HARDWARE EMULATION AND REAL-TIME SIMULATION STRATEGIES FOR THE CONCURRENT DEVELOPMENT OF MICROSATELLITE HARDWARE AND SOFTWARE

by

George James Wells

A thesis submitted in conformity with the requirements for the degree of Master's of Applied Science
Graduate Department of Aerospace Engineering
University of Toronto

© Copyright by George James Wells 2001
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

In small satellite projects on short schedules, there is often insufficient time to develop new hardware and subsequently write software once the hardware is tested and ready. However, emulating the hardware may be useful if the effort involved in doing so is kept to a minimum. The purpose of the emulation should be to act as a substitute for the missing hardware so that flight code can be developed concurrently with the hardware. The use of the real-time development system RT-Lab™ provides a flexible environment to develop flight software early in the development cycle of a small satellite. The degree to which hardware can be emulated is investigated using the development of the attitude control system for the MOST microsatellite as an example. A trade study is presented that indicates when the cost of programming the emulator outweighs the benefits. A level of hardware emulation is recommended that facilitates the early development of flight code.
Acknowledgements

I would first like to thank both my supervisors Dr. Christopher Damaren and Dr. Robert Zee for their constant help and support throughout all my research studies.

I would like to acknowledge Mr. Daniel Foisy and the rest of the MOST engineering team for their aid. I would also like to acknowledge the Natural Sciences and Engineering Research Council (NSERC) and the Canadian Space Agency (CSA) for its generous financial support of my graduate studies.

Finally, I want to thank Karen Chang for her love and support in keeping me motivated to get this document finished before my hair turned grey.
# Table of Contents

ABSTRACT .............................................................................................................. II

ACKNOWLEDGEMENTS .................................................................................... III

LIST OF SYMBOLS ............................................................................................ VI

LIST OF TABLES AND FIGURES ......................................................................... VIII

1. INTRODUCTION ............................................................................................. 1

1.1. LITERATURE REVIEW: SMALL SATELLITE DEVELOPMENT AND ATTITUDE CONTROL ........................................... 1

1.2. LITERATURE REVIEW: HARDWARE-IN-THE-LOOP SIMULATOR USE IN SMALL SATELLITE DEVELOPMENT .... 4


1.3. MOST BACKGROUND ................................................................................. 12

1.4. OBJECTIVES ............................................................................................... 14

2. RT-LAB SIMULATOR SYSTEM DESCRIPTION/DEVELOPMENT ............... 16

2.1. HARDWARE SYSTEM .................................................................................. 16

2.2. SOFTWARE/MODEL DESIGN PHILOSOPHY ........................................... 20

2.3. RT-LAB SIMULATION SYSTEMS WITH MULTIPLE NODES ................... 22

3. MOST ACS FLIGHT CODE SIMULATION DEVELOPMENT ...................... 23

3.1. SIMULATION CONFIGURATION ............................................................... 23

3.1.1. Master SuperBlock .................................................................................. 25

3.1.2. Console SuperBlock .............................................................................. 50

3.2. SIMULATION EXECUTION ........................................................................ 51

3.3. SIMULATION SUMMARY ......................................................................... 53

4. MOST ACS FLIGHT CODE SIMULATION ANALYSIS .................................. 55

4.1. MODEL DEVELOPMENT METHODOLOGY .............................................. 55

4.2. WORK EFFICIENCY TRADE STUDY ON EARLY FLIGHT CODE AND SIMULATOR DEVELOPMENT .................. 56

4.3. FLIGHT CODE DEVELOPMENT LIMITATIONS ....................................... 61

5. ACS FLIGHT CODE ALGORITHMS AND TESTS ....................................... 64

5.1. DETUMBLING ALGORITHM USING MAGNETORQUER ACTUATION .......... 64

5.2. COARSE POINTING ALGORITHM USING REACTION WHEEL ACTUATION .... 64

5.3. REACTION WHEEL DESATURATION ALGORITHM USING MAGNETORQUER ACTUATION ...................................... 68

5.4. ACS SIMULATION TEST RESULTS ............................................................ 69

6. FUTURE EXPANSION OF MOST SIMULATOR ............................................ 71

6.1. MOST COMMAND VERIFICATION FACILITY (ACS PROCESSOR AS HARDWARE-IN-THE-LOOP) ......................... 71

6.1.1. Changes Made To Create MOST CVF Simulator .................................... 73

6.1.2. Note on CVF Development .................................................................... 83

6.2. COMPLETE MICROSATELLITE SIMULATOR ........................................... 84

7. SUMMARY & CONCLUSION ......................................................................... 86

8. REFERENCES ................................................................................................. 88
List of Symbols

$C_1$ rotation matrix about 1-axis
$C_2$ rotation matrix about 2-axis
$C_3$ rotation matrix about 3-axis
$l_1$ 1-axis $[1 \ 0 \ 0]^T$
l_2 2-axis $[0 \ 1 \ 0]^T$
l_3 3-axis $[0 \ 0 \ 1]^T$

$\mathbf{i}_i = \begin{bmatrix} \mathbf{i}_1 & \mathbf{i}_2 & \mathbf{i}_3 \end{bmatrix}$ inertial reference frame

$\mathbf{i}_p = \begin{bmatrix} \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \end{bmatrix}$ perifocal reference frame

$\mathbf{i}_s = \begin{bmatrix} \mathbf{s}_1 & \mathbf{s}_2 & \mathbf{s}_3 \end{bmatrix}$ solar pointing reference frame

$\mathbf{i}_b = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 \end{bmatrix}$ microsatellite body reference frame

$t$ time

$T_s$ simulation time step

$- \frac{d}{dt}$

i orbital inclination
e eccentricity
$\Omega$ ascending node right ascension
$\omega$ argument of perigee
$\theta$ true anomaly
$i_E$ seasonal inclination of Earth
$E$ eccentric anomaly
$M$ mean anomaly

$\hat{r}$ radius of orbit (normalized to a magnitude of 1.0)
r_o radius of orbit
$s$ direction of Sun (normalized to a magnitude of 1.0)
$X$ angle of sunlight striking microsatellite
$\gamma$ angle of Earth's shadow
$\mu_E$ Earth's gravitational constant
$R_E$ radius of Earth
$\alpha$ right ascension of Greenwich
$m_E$ magnetic moment of Earth
$H_0$ dipole strength at surface of Earth
$b$ magnetic field of Earth

$I$ inertia tensor of microsatellite
$\theta$ microsatellite Euler angle states
$\omega$ microsatellite rate states
$h_w$ angular momentum of reaction wheels
$g$ total torque applied on microsatellite
$g_c$ applied reaction wheel control torque
$g_{\text{m}}$ total applied magnetic torque
$g_{\text{mc}}$ applied magnetorquer torque
$g_{\text{md}}$ applied magnetic torque due to natural magnetic moment
$g_{d}$ applied disturbance torque
$S$ kinematical relation matrix

$v$ sensor noise
$R$ sensor noise covariance
$m$ total magnetic moment of microsatellite
$I_w$ reaction wheel moment of inertia
$\omega_w$ reaction wheel rate
$v_w$ applied reaction wheel voltage
d$_{1,2,3}$ reaction wheel dynamic variables
$K_p$ reaction wheel controller proportional constant
$K_i$ reaction wheel controller integral constant
$T_c$ reaction wheel controller time constant
$\zeta_c$ reaction wheel controller damping ratio
$\beta$ commanded reaction wheel slew angle
$X, Y$ sun sensor coordinates
$\theta_{b1}, \theta_{b3}$ sun sensor offset angles
$Q_1, Q_2, Q_3, Q_4$ sun sensor photodiode currents
$A_0, A_x, A_y, B_0, B_x, B_y$ sun sensor calculation constants

$k_b$ detumbling algorithm constant
$b_0$ observed magnetic field
$b_m$ calculated magnetic field rate of change
$T$ kinetic energy
$x$ complete state vector
$\Delta \theta_{bt}$ correction Euler angle states for solar pointing frame
$\Delta \omega_{bt}$ correction rate states for solar pointing frame
$\Delta x_{bt}$ correction state vector for solar pointing frame
$K_p$ coarse pointing command proportional constant
$K_d$ coarse pointing command derivative constant
$K$ complete coarse pointing command constant
$A$ linearized system model
$B$ output feedback matrix
$C$ observer matrix
$L$ observer feedback matrix
$J$ performance function
$Q, R$ performance function weighting constants
$k_m$ desaturation algorithm constant
$\omega_{wd}$ desired reaction wheel angular momenta
$\omega_{d}$ desired reaction wheel rates
List of Tables and Figures

Table 3.1: Simulator Parameters ........................................... 52
Table 3.2: System Modelling Breakdown ................................... 54
Table 4.1: Useful Work Breakdown by System .......................... 58
Table 4.2: Cumulative Simulation Work Analysis ....................... 60
Table 5.1: Magnetic Field Values in Various Orbital Positions ....... 68
Table 6.1: Principal Moments of Inertia of MOST About Centroid .... 73
Table 6.2: Simulation Modification Summary ............................ 74
Table 7.1: Flight Code Development Conclusions ....................... 87
Table 7.2: Microsatellite CVF Conclusions ............................... 87

Figure 1.1: FORTE Hardware Configuration ............................. 5
Figure 1.2: FORTE Closed-Loop Configuration ......................... 5
Figure 1.3: SATSIM: Simulation Software Model ....................... 7
Figure 1.4: Hardware Emulation Interface ............................... 9
Figure 1.5: ISDASE Simulated Subsystems ............................. 10
Figure 1.6: Simulator Hardware Configuration ......................... 12
Figure 1.7: MOST Microsatellite ......................................... 12
Figure 1.8: Continuous Viewing Zone Diagram ......................... 13
Figure 2.1: RT-Lab Computer Configuration ............................ 16
Figure 2.2: RT-Lab Main Window ........................................ 18
Figure 2.3: SystemBuild Simulation Setup Window ................... 19
Figure 2.4: Simulation Top-Level SuperBlock ......................... 21
Figure 2.5: RT-Lab Multi-Node Simulator .............................. 22
Figure 3.1: MOST Systems Diagram ..................................... 23
Figure 3.2: MOST Body Axis Frame and Dimensions .................. 24
Figure 3.3: Order of ACS Sub-System Emulation Creation .......... 24
Figure 3.4: Reaction Wheel – Simulator Wiring Diagram ............ 40
Figure 3.5: Reaction Wheel Serial Packet Format .................... 40
Figure 3.6: Reaction Wheel Moment of Inertia Calculation ........... 43
Figure 3.7: Voltage to Wheel Speed Relationship (Open-Loop Voltage Mode) 43
Figure 3.8: Poor Performance of HW Reaction Wheel at <10 rad/sec. (Black Line) 43
Figure 4.1: Simulation Development Methodology Flowchart ........ 57
Figure 4.2: Simulator Work Breakdown ................................. 57
Figure 4.3: Simulator Interface Diagram ............................... 59
Figure 4.4: Work Efficiency Plot Based on MOST Simulation ........ 59
Figure 4.5: Cumulative Work Inefficiency Plot Based on MOST Simulation 61
Figure 4.6: Work Efficiency Extrapolation ............................ 63
Figure 4.7: Cumulative Work Efficiency Extrapolation ............... 63
Figure 5.1: Destabilization of Coarse Pointing Control ............. 68
Figure 5.2: Detumble Experiment Results ............................. 69
Figure 5.3: Coarse Pointing/Desaturation Experiment Results (Angular Position) 70
Figure 5.4: Coarse Pointing/Desaturation Experiment Results (Reaction Wheel Rates) 70
Figure 6.1: Command Verification Facility Configuration ............ 72
Figure 6.2: Software-Hardware-Hardware-Software Connection SuperBlock 83

viii
1. Introduction

Microsatellite projects tend to have small budgets and short schedules. This places constraints on how much work can be done in the early stage of development. At this stage, some hardware for the microsatellite might not be available because it has yet to be developed. Time spent creating this hardware will delay the development of flight code that requires the presence of this hardware. If the functionality of the hardware can be efficiently emulated using software, then it would be possible to use a computer simulation system to replace the missing hardware. Along with a space environment software model, this would allow the development of flight code while the hardware is being developed. The simulator should be one such that once the hardware is available, it can be inserted into the simulation, replacing its software emulation. The simulation system can then be used to test the interaction between flight code and hardware while working in a simulated space environment. The simulator can also provide operations support for the microsatellite after it is launched and be used to validate upgrades to flight code before they are uploaded to the orbiting microsatellite.

1.1. Literature Review: Small Satellite Development and Attitude Control

Small satellite development is now being recognized as a viable option for performing space science missions. The Jet Propulsion Laboratory has been doing research since 1996 on validating new technologies and project management techniques for use in small satellite projects with short development life cycles [4,5]. These papers focused on microelectronics and optimal design methods. The United States of America is not alone in recognizing the value in small satellite projects. In 1996, the Space Science branch of the Canadian Space Agency (CSA) initiated the Small Payloads Program (SPP). The aim of the program is to encourage Canadian universities and corporations to work together in the development of space science microsatellite projects. The goal is the launching of one microsatellite every 3 years. One of the constraints of the program is that the projects must have a cost no greater than CDN$4M, from the beginning of the mission to one year of orbital operations. The Microvariability and Oscillation of STars (MOST) microsatellite, being developed in part at the Space Flight Laboratory (SFL) of the University of Toronto Institute for Aerospace Studies (UTIAS) was the first project to be
The MOST mission plays a significant role in this thesis because most of the hardware emulation work done is based on its attitude control system.

It is only recently that advanced attitude control was required for microsatellite projects; as the scientific missions for microsatellites became more complex, better control schemes, whether they are earth pointing or inertial pointing, became a necessity. Such advanced control is required for the MOST mission because it must be able to point in a specific direction for weeks on end. In order to do any flight code development using a real-time simulation system, it is necessary to become familiar with the ACS routines required for MOST and the research that has been done on implementing such routines on past small satellite missions.

One attitude control system used by MOST is a set of three magnetorquers accompanied by a three-axis magnetometer. The magnetorquers will primarily be used to detumble the satellite whenever its rates of rotation with respect to its inertial pointing frame exceed 2 deg/s. Michele Grassi [8,9,10,11] is an expert in the use of fully magnetic control schemes for small satellite missions. He and his colleagues have developed and tested magnetic control schemes that can be used for all the attitude control functions of a small satellite, including pointing routines. Though the research focused on the control schemes for detumbling a microsatellite, much was learned in general on magnetic attitude control. Rafal Wisniewski of Aalborg University (Denmark) has also done research on the use of magnetic attitude control by small spacecraft in near polar orbits subject to gravity gradient torque [12,13].

In order to point in a specific direction, MOST uses a set of three reaction wheels\(^1\). State estimation is done using an on-board orbit propagator and using a Kalman filter on the sensor readings from the magnetometer, a sun-sensor, and rate sensors that are part of the reaction wheel package. Due to limitations in processor memory and speed, the state estimation scheme for MOST must not be too complex. Kalman filtering is considered in [14,15]. The second one, an evaluation paper done by Dr. Chris Damaren of UTIAS for Dynacon Enterprises Ltd., was the primary source for the coarse pointing scheme developed using the simulator system.

\(^1\) A Dynacon Enterprises Ltd. Miniature Reaction Wheel or "Microwheel".
While in coarse pointing mode, it will be necessary to manage the momentum of the reaction wheels using the magnetorquers so that the wheels do not approach saturation speed. Work by Xiao-jiang Chen and Willem Steyn of the University of Surrey present an excellent summary of numerous reaction wheel desaturation routines [16,17]. They compare the standard cross-product control law with two LQR optimized controllers and a minimum energy controller. They also studied reaction wheel desaturation using only thrusters, only magnetorquers, and both together. When it comes to performing reaction wheel desaturation, it becomes apparent that there is no precise technique available to determine the control gains required for the magnetorquers. The position control of the microsatellite can become unstable while the wheels lose their momentum if the gains are too high. Hari Hablani of Rockwell International developed a pole-placement technique that can be used to correlate control gains with close-loop pole locations [18]. This allows for more efficient desaturation routines as the power consumed by the magnetorquers will be reduced, as well as preventing the onset of instability in position control.

MOST will have a “star-tracker” CCD system so it can perform fine pointing attitude control. Some research was done [19] on the functionality of “star-trackers”, however the “star-tracker” was never emulated using the simulator system and no flight code was written dealing with fine pointing attitude control. This could be a future feature added to the simulator system. However, the emulation will be very complex and might be beyond the capabilities of the system to handle. A more reasonable approach might be to have the “star-tracker” software running on a separate computer and link it to the simulation as hardware-in-the-loop via a serial connection, in effect adding it in as a slave node.

Along with this specific research done on small satellite ACS, general ACS concepts are dealt within [20,21]. Though none of them are used by MOST, they could be incorporated in the future on the simulation system to test their effectiveness for future microsatellite missions. The first paper discussed a minimum power optimal control scheme for the Scientific Microsatellite for Advanced Research and Technology (SMART) microsatellite being developed, in part, by Michele Grassi. Reducing power usage on microsatellites is critical because they tend to not have much available power due to their small size and mass. The second paper dealt with the
need for an autonomous orbit maintenance system so that the specific orbit of the small satellite is known. Such a maintenance system would help reduce operations costs because mission planning can be done far in advance without the need to update the orbit model to account for perturbations. Orbit maintenance is also useful for maintaining the positions of a constellation of small satellites.

The development of the ACS of MOST can be compared to a past microsatellite mission called CATSAT [22,23]. Though more massive than MOST at 140 kg., it was to be placed (in 1999) in a similar orbit (Sun-synchronous) and its primary mission was also astronomical in nature - the study the X-ray and gamma-ray spectra of gamma-ray bursts. CATSAT is the result of the collaboration of students and professors from several universities and has similar ACS requirements as MOST.

Many details concerning MOST are given in a series of SFL and Dynacon internal reports. The reports deal with the technical details of all the sensors and actuators of MOST, especially the Dynacon “Microwheel”, as well as the on-board computer (OBC) configuration of MOST and the communication protocols used by the OBC buses. Some of this information is summarized in Appendix B.

1.2. Literature Review: Hardware-in-the-loop Simulator Use in Small Satellite Development

Given the requirements for developing effective ACS flight code similar to what will be used on for MOST, we now consider previous usage of real-time hardware-in-the-loop simulators for past microsatellite missions. The use of a hardware-in-the-loop simulator involving the emulation of hardware is not new in small satellite development. It is important to note that though many microsatellite projects use computers to simulate their ACS systems, these computer simulations use only software and do not include the ability to link actual satellite hardware with the simulator.

Past real-time hardware-in-the-loop simulation work has been done at Los Alamos National Laboratory [1], Utah State University [2], and the Harbin Institute of Technology in
All three institutes used computer simulator systems that combined both commercial off-the-shelf (COTS) technology with in-house developed systems, all three involved hardware-in-the-loop, and all three used their simulator to design and test small satellite systems.

1.2.1. Los Alamos National Laboratory: FORTE Hardware-in-the-Loop Simulation [1]

In order to effectively develop the attitude control algorithms for the Fast On-Orbit Recording of Transient Events (FORTE) small satellite, Kimberly K. Ruud, Hugh S. Murray and Troy K. Moore used a PC based (120 MHz Pentium) simulator system developed by Ithaco Inc. and Los Alamos National Laboratory. The hardware-in-the-loop simulation system simulated the dynamic performance of a satellite in orbital space, including such disturbance torques as gravity gradient, aerodynamic drag, solar radiation pressure and residual magnetic dipole moment.

Figure 1.1 shows the hardware configuration of the attitude control and determination system (ACDS) of FORTE. Though it is easy to test the functionality of each individual piece of hardware, “[testing] of the flight ACDS systems and control algorithms is very limited without the simulation.” To accurately test the algorithms, ... [ACDS] data need to correspond to a valid spacecraft attitude and orbital location [1].” The PC simulation was designed to work in two modes. In open-loop mode, all the attitude control laws are implemented on the PC with no hardware connected to the simulator. The authors primarily used the simulation in its second mode: closed-loop. In this mode, as shown in Figure 1.2, the spacecraft flight computer and the

---

**Figure 1.1:** FORTE Hardware Configuration [1]

**Figure 1.2:** FORTE Closed-Loop Configuration [1]
data acquisition card (DAC) are connected as hardware-in-the-loop to the simulation PC via a custom-made interface electronics box used to buffer and condition the signals. The attitude control laws are implemented on the flight computer and the simulation replaces the flight hardware on the left side of Figure 1.1. All simulated sensor data collected by the PC is sent to the flight computer via the DAC and all actuator commands from the flight computer are sent back to the PC where the spacecraft response is simulated.

The simulator was used to refine the control algorithms and sequences used by FORTE. Scenarios that were studied included separation, acquisition on orbit, control system parameter sensitivity studies, sensor noise simulations, antenna deployment, and momentum desaturation. The simulation allowed a thorough testing of all these scenarios using different attitude control algorithm configurations in a variety of space environments. This facilitated the final development of previously written attitude control code and allowed the authors to refine and optimize the position control capabilities of the spacecraft.

1.2.2. Utah State University: An Integrated Development System for Small Satellite Attitude Control Systems [2]

The Space Dynamics Laboratory (SDL) of Utah State University developed an integrated system to design and test attitude control systems for small satellites. The authors determined that though the development costs for small satellite ACS systems differed little from full scale projects, the resources available are considerably less. This necessitated the creation of hardware and software simulation tools that could be efficiently used in developing small satellite systems. The simulation tools could also be used for educational purposes at the university. The system comprised of 5 tools: dynamic simulation software, an air bearing table, the hardware emulator electrical interface, graphical and data handling software, and real-time display software.

Figure 1.3 shows the dynamic simulator software model. The model, called SATSIM, was developed for UNIX-based machines and is partially written in both Fortran and C. It comprises a numerical integrator with a series of software modules that model the dynamics of
the spacecraft, the environment, and the sensors and actuators of a small satellite. During the initial development of an ACS system, as hardware is selected, the modules are refined to include accurate software simulations of the sensors, actuators, and the I/O interface. After this point, the simulator can be used to develop flight control software. After the software is written, the controller code can be evaluated by testing it on ACS hardware running on the air-bearing table, or by executing it on the hardware emulator interface, which provides and accurate model of the electronic response of the sensors and actuators to the controllers commands. When the actual small satellite is in orbit, SATSIM can be used during operations to verify the response of the spacecraft to command sequences before they are uploaded. Using SATSIM to write the flight code proved to be very effective. The code could be easily tested during development because it was linked with a dynamic model providing realistic stimuli and responses.

Using the air bearing table to test ACS systems was not always feasible. This was especially true for small satellite projects that required high precision pointing accuracy. However, SDL does have the necessary equipment to use the air bearing table inside a chamber that uses ground support equipment to model the sun and Earth. Future upgrades include the use of a three-axis Helmholtz coil chamber to allow the use of magnetorquer and magnetometer hardware. If the air bearing table is not the appropriate tool to use for testing the flight code, the
hardware emulation interface can be used. Figure 1.4 is a diagram of the hardware emulation interface. It was built using a Pentium 166 MHz PC and appropriate I/O boards. The emulation software was written in both Fortran and C. The various I/O lines can be used to simulate the data transmission of numerous sensors (e.g. sun sensors, magnetometers) and actuators (e.g., magnetorquers, cold-gas thruster systems). The spacecraft dynamic and environment model is a slightly modified version of the SATSIM model. As the hardware emulation interface was operated, a real-time display of collected data was available. After a simulation run, graphical and data handling software was available. Data could be plotted, scaled, merged with other test results, and placed into a MATLAB compatible format for further analysis.

SDL used their simulation tools for the four stages they identified in the typical development cycle of an ACS design for small satellites: 1) Conceptual Planning, 2) Design and Development, 3) Testing, 4) Operations. Conceptual Planning involved determining the necessary control requirements and choice of actuators and sensors. The dynamic simulator was used to create simple models which generate initial estimates of capabilities of the ACS design and to perform tradeoff studies. During the Design and Development stage, the ACS flight code is written. By using a more detailed SATSIM model, flight code can be written and tested against emulated actuators, sensors, and dynamic environment models. It was found that this “write-then-test” sequence reduced the development time of the flight code. The SATSIM package was also used to test previously written flight code of other missions. The testing phase involved using either the hardware emulation interface or the air bearing table.
The emulator was used to evaluate the flight code, the controller electronics, and the electronic interfacing. The air bearing table was used to evaluate the actual flight hardware, which would interact with the ACS flight code and dynamic models running on SATSIM. Though useful for functionality checks, high fidelity replication of the space dynamics was impossible, hence the use of software simulation in the first place. The Operations stage occurred after the small satellite is launched. When used as ground-based support, ACS command tasks can be verified using the simulation code before they are uploaded to the actual satellite.

Finally, SDL noted the tradeoff that exists between developing a custom simulation system and purchasing commercial hardware/software. On one hand, purchasing commercial code reduces development time and the effort to maintain the software. On the other hand, having intimate knowledge of the details of your own written code can provide you with additional capabilities and insight.
1.2.3. Harbin Institute of Technology (HIT) (China): The Integrated System for Design, Analysis, System Simulation and Evaluation of the Small Satellite [3]

The paper began by describing the growing interest in using real-time hardware-in-the-loop simulation as part of an integrated conception and design approach in developing small satellites. Such a simulation system was developed at HIT: the integrated system for design, analysis, system simulation and evaluation of small satellites (ISDASE). It can be used "to optimize, simulate and evaluate the system scheme during the conception design stage, to demonstrate and verify the performance and specification of the components and subsystems during the development stage, and to deal with fault diagnosis and procession during the test and operation stage [3]."

ISDASE consisted of a Pentium 200 MHz PC using MatrixX/SystemBuild 6.0 to design and control the simulation. This was connected via a PC LAN to a single-axis air bearing table and a real-time simulator (AC104) used to set up the research and test platform. The systems that were designed on the PC and included as part of the simulation go beyond just the ACS. Figure 1.5 shows the subsystems that were part of the simulation and the connectivity between them. The main function of the research and test platform was to link flight hardware to ISDASE as hardware-in-the-loop. It can also be used to evaluate software components and subsystems.

![ISDASE Simulated Subsystems](image)

The research and test platform was used in two configurations:
1. **Computer-in-the-loop Simulation**: Except for the on-board computers, all other hardware was emulated using software. These emulations, along with the space dynamics and environment model, are run on the AC104 real-time simulator. The modules can be linked to the on-board computers via numerous interfaces, such as A/D, D/A, D/D, and serial. This configuration was used to check the on-board programs and to test this flight code on the ground.

2. **Hardware-in-the-loop Simulation**: Similar to the previous configuration, except that some of the software emulated systems are substituted for the real flight hardware. This configuration was used to check the performance indices of the hardware, to validate the software emulation of the hardware, and to deal with fault diagnosis.

ISDASE was used to design HITSAT-1, the first small satellite developed at HIT. The ISDASE setup for the small satellite included a gyro, reaction wheel, and two on-board computers (ACS and House Keeping (HK)) as hardware-in-the-loop. The gyro and reaction wheels were mounted on a single-axis air bearing table. The housekeeping module was designed using MatrixX 6.0 and compiled using AutoCode (see Chapter 2) and formatted to run on the AC104. The ACS flight code was uploaded to both the ACS computer and the HK computer, as a backup. The AC104 is linked to the on-board computers via an RS-232 serial interface and to a simulated ground station via ethernet. The system configuration is shown in Figure 1.6.

HIT was pleased with ISDASE and found it to be a useful tool for use in the development of a small satellite mission. It possessed the following beneficial characteristics: an advanced simulation platform (MatrixX 6.0), topological configuration (modular, easy to design and replace emulation and mathematical models), complete hardware interfaces (serial, digital, analog), convenient windows interface (results displayed graphically).
1.3. MOST Background

The Microvariability and Oscillation of STars (MOST) microsatellite (Figure 1.7), being built in part at Dynacon Enterprises Limited, the University of Toronto Institute for Aerospace Studies Space Flight Laboratory, and the University of British Columbia, will be Canada’s first space telescope [4,5]. Being developed under the Canadian Space Agency’s Small Payloads Program, it is scheduled for launch in 2002 and will conduct long-duration photometry of nearby stars.

All stars oscillate in luminosity over their entire lifetime. Even our own Sun experiences these oscillations, though the amplitude of the oscillation is not as severe as those found in variable stars. These oscillations give an indication of the age of the star, and the study of the
oscillations of nearby stars would give an indication of their apparent ages and thus set a lower limit on the age of the universe. However, these oscillations are extremely difficult to measure from the ground due to atmospheric distortion. Hundreds of kilometers above the surface of Earth in a sun-synchronous orbit, a satellite could be used to perform this long-duration astronomy. It can take many weeks to collect enough oscillation data for just one star.

Figure 1.8: Continuous Viewing Zone Diagram

MOST requires an accurate three-axis attitude control system in order to successfully complete its mission. It must be able to hold its position within half and arcminute for days on end in order for the on-board CCD camera to collect enough data on the luminosity of nearby stars. The ACS system is comprised of three reaction wheels, three magnetorquers, a three-axis magnetometer, a two-axis sun-sensor, and three rate sensors, which are included with the reaction wheels. The ACS processor board is based on the Motorola 56303 digital signal processor (DSP) and has a backup, kept in cold storage, that is only used if the primary DSP fails.

MOST will be placed into a dawn-dusk sun-synchronous orbit at around 785 km altitude. This will allow it to have a large continuous viewing zone (CVZ). The CVZ is the section of the sky that MOST can continuously view for up to seven weeks (see Figure 1.8). The anti-solar CVZ faces directly away from the Sun and is the zone where MOST can look for candidate stars to study. The dawn-dusk sun-synchronous orbit is also ideal from the point of view of power.
generation and for using a sun sensor for ACS operations. At 785 km, MOST will only experience eclipses during 3 months in the summer. These eclipses will only last a maximum of just over 17 minutes per 100 minute orbit.

1.4. Objectives

If real-time, hardware-in-the-loop simulation systems are going to be frequently used when designing small satellites, it will be necessary to identify strategies that can be employed so that work can be done in an expedient manner. Being able to get a simulator system working quickly and being able to emulate missing hardware with little effort is critical in order to develop flight code at an early stage, concurrent with the development of the missing hardware. The goal is to minimize any “throw-away” work; work that cannot be used either on the microsatellite or by the simulation system when it is used as ground support equipment for the satellite program.

The following steps detail the procedure that will be taken to determine and develop some of these strategies that can be used to perform efficient and concurrent microsatellite software and hardware development.

1. Purchase a simulator system that combines both real-time hardware-in-the-loop simulation with easy-to-use software so that emulations of missing hardware can be made with as little effort and coding as possible.

2. Create a model of the Attitude Control System (ACS) of the MOST microsatellite that can be executed on the simulator system. This includes software emulations of the ACS hardware as well as writing sample ACS flight code. Assume a maximum development time of 10 months, approximately how long it will take to develop the prototype ACS processor board. Keep track of the amount of work, in terms of time spent, that goes into emulating missing hardware systems and developing flight code that can run on the ACS processor once it is ready. Any code written in under 10 months is time that the simulator saved in code development after the ACS processor is available. The developed sample code should duplicate, as much as possible, the same functionality performed by the actual ACS flight code used on MOST.
3. In performing Step 2, it should be demonstrated that the ACS flight code developed is functional and appropriate for use by a microsatellite. Control and state estimation algorithms for detumbling and coarse pointing are required. The algorithms must take up little memory space and cannot place great power demands on the actuators.

4. Using the experience gained from this development, create a methodology that can be used when doing work on the simulation system so that "throw-away" work is minimized. Based on this methodology, a trade study will be done on the work performed on the MOST ACS simulation to determine any relationship between the efficiency of the work done for each ACS subsystem and the complexity of the subsystem.

5. Based on the results of the trade study, determine the types of flight code that can be written early in the life of a microsatellite project vs. the flight code that should not be developed until the hardware is available.

6. Modify the simulation model by removing the "throw-away" work and preparing it so that the ACS processor can be included as hardware-in-the-loop. The flight code developed previously using the simulator can be used by the ACS processor. If this can be done efficiently, the simulator can then be used to test the ACS processor before it is launched. After the microsatellite is launched, this simulator system can also be used to test any ACS routines or modifications to the flight code before they are uploaded to the microsatellite for execution.

The following chapters of this document summarize the work done in performing this procedure and highlight all of the strategies developed. Conclusions were then drawn on how best to use a real-time simulator system to do concurrent software and hardware work on a microsatellite project. An analysis was also done on the sample ACS algorithms created using the simulator.
2. RT-Lab Simulator System Description/Development

2.1. Hardware System

A simulator system, known as RT-Lab\textsuperscript{TM}, made up of COTS hardware and software components was purchased to minimize the work needed to develop the system. RT-Lab is a product of the Montreal based company, Opal-RT. Opal-RT specializes in creating real-time simulation systems for engineering design applications [24]. See Appendix A for more information on the company.

The RT-Lab system is a multiprocessor platform that enables real-time simulation of complex models. The system has easy-to-use software that can be used for hardware emulation. The system also includes software that creates, executes, and controls the real-time simulation. The system used here consisted of two Pentium II 400 MHz computers. Details on the computer hardware can be found in Appendix A. Except for specialized hardware interfaces in one of the computers, the computers are similar to a typical desktop machine. The configuration of the multi-processor system is shown in Figure 2.1. This simulation system is the simplest one that can be purchased from Opal-RT. Simulation systems involving more than two computers can be purchased. Such configurations are used to simulate highly complex engineering designs (see Section 2.3).

The first computer is the host computer of the RT-Lab system. It has Windows NT as its operating system and runs the RT-Lab software. It is from this machine that the user creates the model that will be simulated in real-time. The host machine is also used to display and store data collected during the simulation run. The user can also interact with the model on the host machine by giving it input either before the simulation is started or while the simulation is running.

\footnote{RT-Lab is a trademark of Opal-RT.}
The software used to create the simulation model is MatrixX/SystemBuild™. SystemBuild is a control block mathematical program. By using built-in mathematical function blocks and user-designed code blocks written in C, the user can design a state model of a system. Past aerospace-related model development on SystemBuild include aircraft, spacecraft, and robotic systems. On the RT-Lab system, model work can also be done using Matlab/Simulink™, a program very similar to SystemBuild.

The second computer in the RT-Lab distributed system is the target computer. This machine runs the QNX operating system. QNX is a version of UNIX that specializes in real-time computation. After a model is designed in SystemBuild on the NT host, it is simulated on the QNX machine to take advantage of the real-time kernel, timers, and interrupts that are available. Appendix A has more information about the QNX OS and the timers that it uses to guarantee real-time response. These real-time tools make for an accurate simulation test bed because they guarantee a response to an interrupt within 1 μs, much faster than conventional operating systems. The target computer is also used to link hardware with the simulation. Using the motherboard slots on the target computer, PCI and AGP boards can be connected to provide a variety of data communication interfaces. The serial ports on the computer can also be used as a data interface. These interfaces are used to connect hardware systems to the simulator. This hardware reacts to the model in every way, providing both input and reacting to the simulation as necessary. The simulator used here has interfaces for serial communication - RS-232 and RS-422/485 formats, and digital/analog IO (see Appendix A for more details). All of the software drivers are already installed on the Target node. All the user has to do is use the custom-made SystemBuild blocks to run these drivers, linking the model to the hardware interfaces. For RT-Lab systems that have multiple target computers, you can configure the simulation so that each computer handles one aspect of the model being simulated. For the purposes of this simulation, the target computer handled that entire model.

The RT-Lab software running on the Host machine provides an easy-to-use interface that can be used to perform all the necessary functions to edit, compile, and run the simulation. Figure

---

3 MatrixX/SystemBuild is a trademark of Integrated Systems Inc. (ISI)
4 Matlab/Simulink is a trademark of MathWorks.
2.2 is the top-level window of the RT-Lab software. Once a SystemBuild model is designed on the Host computer, it can be loaded using the "Open Model" button on RT-Lab. RT-Lab can then be used to open a SystemBuild window of the model with the "Edit" button. With the model loaded, the "Configuration" button is used to prepare the real-time simulation. Features included are global variable declarations, a debug execution mode, and the ability to include C program files written by the user for inclusion with the SystemBuild model. This C code must be placed in the SystemBuild simulation model in the form of UserCode blocks. Once the design of the simulation model is finished, the "Compile" button is used to prepare it for execution. For compilation, RT-Lab uses AutoCode, another ISI software program, to convert the SystemBuild model into C code. This code, along with any C code written by the user to be included with the simulation is then transferred by RT-Lab to the QNX computer via an ethernet connection. It is then compiled using WATCOM C v5.1 and readied for execution.

Figure 2.2: RT-Lab Main Window

After compilation, the "Probe Control" button is then used to control the number of data point measurements made per time step, the length of the time step being set in SystemBuild. The slide in bottom-right corner of the RT-Lab window is used to control the type of simulation and the speed of execution. If the simulation is placed in "Simulation" mode, RT-Lab will execute the
simulation as fast as the Target computer will allow. Though this mode takes advantage of the QNX real-time kernel, some data will be lost during the execution. If the simulation is placed in "Software Synch." mode, then the simulation will run in real-time taking full advantage of the QNX real-time kernel. The speed of simulation can be controlled using the “Calculation Step” slide. If set to a value of “1”, one second of simulation time equals one second of actual time. One other simulation mode available, not used in this research, is "Hardware Synch." mode, where the execution of the simulation is based off a hardware timer connection to the simulator as hardware-in-the-loop.

Figure 2.3: SystemBuild Simulation Setup Window

Once the simulation has been prepared, the "Load" button is used to initialize the simulation on both the Target and Host computers. The compiled C code on the Target computer is initialized and a console interface, created by the user when the model was designed using SystemBuild, is initialized on the Host machine. This console allows the user to interact with the simulation during
its execution. After the SystemBuild console is opened, it has its own set of command buttons including a "Simulate" button (see Figure 2.3). This button allows the user to define the time vector of the simulation. The user must also select the "Interactive" option so that the user can interact with the simulation during its execution. Before selecting “Ok” on this SystemBuild window, the user can select what integration numerical method to use for any state space calculations done by the simulation. Once this is done, the simulation can then be controlled using the "Execute" and "Pause" buttons found on the RT-Lab interface. When the user wishes to stop the simulation, the user must close the console window and push the "Reset" button on the RT-Lab interface.

2.2. Software/Model Design Philosophy

In SystemBuild, the primary block used to construct a simulation model is called a SuperBlock. SuperBlocks are used to contain all the other types of system blocks found in SystemBuild, including C UserCode blocks and other SuperBlocks. In effect, a simulation model is made up of layers of embedded SuperBlocks, each one encompassing the functionality of the various subsystems of the simulation. For an RT-Lab simulation, the top level SuperBlock (Figure 2.4) contains two other SuperBlocks: the Master block and the Console block. The Master block contains all the model blocks pertaining to the actual simulation. Everything that is to be executed on the QNX target computer is placed here. The Console block contains all of the tools (eg. buttons, slides, graphical displays) the user wants for the console interface used to send and display simulation data on the host computer while the simulation is running.

A constraint on the model design is that these two blocks must be included so that the model can work with the RT-Lab system. When the RT-Lab software converts the model into C code and sends it to the target node, it looks for the Master block to know which blocks are to be used in the simulation. It also creates the console interface using the information in the Console block. As mentioned previously, the console interface is employed by the user on the Host computer to interact with the simulation running on the target computer. Another constraint is that these two SuperBlocks must be set to Discrete instead of Continuous in their Properties window. The discrete time step selected will be the base time step of the RT-Lab simulation. Every other SuperBlock
inside the Master and Console SuperBlocks can be set to different discrete time steps or even set to Continuous if so desired.

**Figure 2.4: Simulation Top-Level SuperBlock**

<table>
<thead>
<tr>
<th>Discrete SuperBlock</th>
<th>Sample Period</th>
<th>Sample Skew</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Enable Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA Overall</td>
<td>0.1</td>
<td>0.0</td>
<td>0</td>
<td>9</td>
<td>Parent</td>
</tr>
</tbody>
</table>

Another constraint is that no C commands or functions can be used that postdate either WATCOM or MS C v5.1. The WATCOM compiler that comes with the simulator will not compile them. However, this is a good constraint to have in place. Most microsatellite processor hardware will be simple digital signal processors (DSPs) or low end CPUs (eg. 80186) and are limited in their capability to handling complex C algorithms. By forcing this constraint now, it will make porting flight code developed on the simulator to the processor hardware, when it is available, easier.

When designing a model, the strategy is to maximize the use of built-in SystemBuild mathematical blocks whenever possible for sections of the simulation that are emulating missing hardware (eg. sensors, actuators). At the same time, the use of C code user blocks is maximized for systems requiring flight code development (eg. ACS processor). Though not done in this project, the C code written in these user code blocks could be used, with minor modifications, on the microsatellite itself. The ultimate goal is to reduce the amount of time spent emulating missing hardware while still generating usable flight code early in a microsatellite project.
2.3. RT-Lab Simulation Systems with Multiple Nodes

Figure 2.5 is a diagram of a multiple node simulator system. Each of the nodes is a real-time Target computer that can be set to run part of the entire simulation. To accomplish this, the user must create Slave SuperBlocks, one for each Target node, along with the Master and Console SuperBlocks. The simulation functionality, both C UserCode blocks and regular SystemBuild blocks, encompassed in each Slave SuperBlock will be transferred to its proper Target node and compiled separately. This distributes the computational burden of running a complex simulation in real-time to multiple processors. The Spacecraft Dynamics Lab at UTIAS purchased a multiple Target node simulation system. This simulator, along with its purpose, is described in Section 6.2.
3. MOST ACS Flight Code Simulation Development

3.1. Simulation Configuration

The simulation model designed on the RT-Lab system was based on the ACS of the MOST microsatellite. Figure 3.1 is a systems diagram of MOST. The shaded boxes indicate which systems were included as part of the simulation model. The simulation model focused on the primary ACS processor and its peripherals. The systems modeled include a full set of actuators (three reaction wheels and three magnetorquers) and a full sensor package (three-axis magnetometer, three rate sensors and two-axis sun sensor). The model also includes an orbital environment simulation, complete with both a dynamic model and a model of the Earth’s magnetic field.

Figure 3.2 shows the body frame axis (referred to as \([x \ y \ z]^T\) or \([1 \ 2 \ 3]^T\)) orientation of MOST. This frame was used in designing the simulation. For the simulation, it was assumed that the body axis frame and the principal axis frame of MOST were aligned exactly so that the moment of inertia matrix \((I)\) would be diagonal. It will be shown in Section 6.1.1 that this assumption was acceptable for MOST.

*Figure 3.1: MOST Systems Diagram*
The MOST ACS simulation model was built one subsystem at a time, starting with subsystems that were simple to model using SystemBuild. Figure 3.3 indicates the order in which the ACS subsystem emulations were created. Note that all the sensors were created first, followed by all the actuators. The orbital environment model and ACS processor (with all its flight code) were the most complex subsystems, requiring many lines of C code, so they were not created until the very end. As the model progressed from its initial configuration to its current state, sample ACS flight code was developed.

The following is a description of the contents of every SuperBlock in the final version of the MOST ACS simulation model. Many revisions were made to the model throughout its creation. They were done to refine the speed of the model and to eliminate bugs as they were

**Figure 3.3: Order of ACS Sub-System Emulation Creation**

- Magnetometer
- Rate Sensors (x3)
- Sun Sensor
- Magnetorquers (x3)
- Reaction Wheels (x3)
- Orbital Environment
- Sample ACS Flight Code
found. The description starts with the Master SuperBlock, proceeding through all the layers of embedded SuperBlocks, and finishes with the contents of the Console SuperBlock.

Diagrams of each SuperBlock can be found in Section I of Appendix C.

3.1.1. Master SuperBlock

The simulation was divided into two primary subsections: the orbital environment model and the microsatellite systems. Each was placed in its own SuperBlock. One other block found in the Master SuperBlock was a Gain block used to convert the attitude and rates of the microsatellite from radians to degrees for display in the Console.

The block found on the left-hand side was the OpComm block. All inputs into the Master SuperBlock had to pass through the OpComm block, with the block outputting the inputs unchanged. This block was used by RT-Lab to identify where the Master subsection of the simulation began. This subsection is the section of the model that ran on the QNX Target node of the simulator. An OpComm block was also placed in the Console SuperBlock, indicating the end of the Master subsection of the simulation.

All of the inputs into the Master SuperBlock (see Figure 2.4) are fed back from the Console SuperBlock. Many of these inputs are commands controlled by the user from the Console interface. These commands were used to place the microsatellite in various control modes. Three of the inputs from the Console were user-controlled disturbance torques, used to test the robustness of the control algorithms. The remaining inputs were simulation variables that had to be passed all the way though the model, from the Master SuperBlock to the Console SuperBlock and back again, in order to eliminate circular algebraic logic errors in the simulation.
The environment model was divided into two important sub-sections, the orbital
dynamics model and the attitude dynamics model. The variables fed from the Attitude
SuperBlock into the Orbit SuperBlock were the components of the rotation matrix defining the
change from the inertial frame to the body frame of MOST. The variables outputted from the
Environment SuperBlock include: the magnetic field in the body frame, the direction of the Sun
with respect to the x-axis face of the microsatellite, the rates of rotation around each body axis,
the true anomaly (θ) of the orbit, the inertial coordinates of the orbit (normalized to a magnitude
of 1.0), and the angles of rotation of the body axis with respect to the inertial frame.

The first three sets of outputs were used by the sensor emulations to generate
environment data they could detect. Two sets of outputs were only used to display the position
and attitude of the microsatellite to the user in the Console interface. They were never used by
the simulation to help the ACS processor estimate the attitude states of the microsatellite.

Two other outputs from this SuperBlock were the true anomaly and magnetic field
values, which were used indirectly by the ACS processor to help generate its own on-board orbit
propagator for use in state estimation. Rather than running a separate but similar orbital model
simulation to be used for attitude control by the spacecraft, the following method was used.
Those orbital environment outputs required by the ACS were multiplied by an error factor to
simulate the difference between the actual orbital position of the spacecraft and its software
orbital model.

The Orbital SuperBlock consisted of three models: the orbital dynamics model, the
magnetic field model, and the Sun position model. All three were created using C UserCode
Blocks. The ramp block was used to keep track of the current simulation run time. This run
time was required by the environment models in order to calculate the orbital position of the microsatellite. The other time block was there to allow the user to have a simulation start time later in the year than the Vernal Equinox (Mar. 21st) at midnight, which was chosen to be the initial starting time.

The orbital dynamics UserCode block required the user to enter the initial orbital position of the satellite using the six Keplerian coordinates (perigee altitude, apogee altitude, $i$, $\Omega$, $\omega$, $\theta$). These coordinates had to be entered by the user before the simulation was compiled. They were entered using the Real Parameters list of the C UserCode block, which was accessed using the Block Parameters button.

The inertial frame $\mathbf{\tilde{i}}_i = [\mathbf{i}_1 \mathbf{i}_2 \mathbf{i}_3]$ had the $\mathbf{i}_1$ axis pointing towards the constellation Pisces, which meant that during the Vernal Equinox, it pointed at the Sun. The $\mathbf{i}_3$ axis pointed in the direction of Earth's North Pole. The perifocal frame $\mathbf{\tilde{p}}_p = [\mathbf{p}_1 \mathbf{p}_2 \mathbf{p}_3]$ had the $\mathbf{p}_1$ axis pointed along the direction of perigee and the $\mathbf{p}_3$ pointed along the direction of the angular momentum of the orbit (i.e., normal to the orbit). The rotation matrix linking the perifocal frame to the inertial frame was

$$C_{ip} = (C_3(\omega)C_1(i)C_3(\Omega))^T$$

$$= \begin{bmatrix} 
    c(\Omega)c(\omega) - s(\Omega)c(i) s(\omega) & -c(\Omega)s(\omega) - s(\Omega)c(i) c(\omega) & s(\Omega)s(i) \\
    s(\Omega)c(\omega) + c(\Omega)c(i)s(\omega) & -s(\Omega)s(\omega) + c(\Omega)c(i)c(\omega) & -c(\Omega)s(i) \\
    s(i)c(\omega) & s(i)s(\omega) & c(i) 
\end{bmatrix}$$

The initial eccentric and mean anomalies were then calculated by the simulation according to

$$E_0 = \tan^{-1}\left(\frac{\sqrt{1 - e^2} \sin(\theta_0)}{e \cos(\theta_0)}\right)$$

$$M_0 = E_0 - e \sin(E_0)$$

As the simulation advanced through each time step, the mean anomaly was recalculated and Newton’s Method was used to determine the eccentric anomaly (using the current mean anomaly
as the initial guess). Here, \( \mu_E \) is Earth’s gravitational constant and the value “a” is the semi-major axis.

\[
M = M_0 + \frac{\mu_E}{a^3}
\]

Newton’s Method Function:

\[
E_{i+1} = E_i - \frac{E_i - e \sin(E_i) - M}{1 - e \cos(E_i)}
\]

From this, the current true anomaly was calculated along with the orbital position and velocity in Cartesian coordinates. This orbit propagator was provided by Dynacon and its functionality was verified on the RT-Lab simulator by running it through one orbit. The outputs from this block were the true anomaly, the radius of the orbit, the position coordinates normalized to a magnitude of 1.0, and the rotation matrix from the solar pointing frame to the inertial frame. The solar pointing frame, \( \mathbf{\tilde{S}}_i = \begin{bmatrix} \mathbf{\tilde{i}}_i & \mathbf{\tilde{j}}_i & \mathbf{\tilde{k}}_i \end{bmatrix} \), had the \( \mathbf{\tilde{i}}_i \) axis always pointing at the Sun, and hence the \( -\mathbf{\tilde{k}}_i \) axis always pointed in the direction of the anti-solar continuous viewing zone (CVZ – see Section 1.3). The \( \mathbf{\tilde{j}}_i \) axis pointed in the direction of Earth’s North Pole. Therefore, the solar pointing frame matched the inertial frame during the Vernal Equinox and rotated about the \( \mathbf{\tilde{i}}_i \) axis with a period of 365.25 days. The rotation matrix from this frame to the inertial frame had the following form

\[
C_{\mathbf{\tilde{i}i}} = \begin{bmatrix}
\cos(\omega_y t) & \sin(\omega_y t) & 0 \\
-\sin(\omega_y t) & \cos(\omega_y t) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

where \( \omega_y \) was the orbital frequency of the Earth around the sun. The solar pointing frame was the nominal direction to point the spacecraft. This kept the telescope pointing along the anti-solar CVZ and in the direction of candidate target stars. It also kept the sun sensor pointing towards the sun so that the spacecraft would have enough sensor information to maintain control of its attitude. The MOST spacecraft will point off this frame using its fine pointing ACS algorithm when a target star is chosen and go into an inertial pointing mode. However, this simulation did not cover this functionality (no star tracker emulation was made) and thus the sample ACS flight code only used this frame to point the spacecraft.
This was a simplified orbital model because the orbit will never decay or become perturbed. However, the focus of the simulation was to perform attitude control experiments to perfect ACS flight code, not study long duration orbit maintenance. Therefore, this simplification did not affect the sample ACS code development.

The model of the Earth’s magnetic field in its geographical frame was a simple dipole model.

\[
\begin{align*}
\text{Magnetic Pole Position: } & \text{Latitude } = 78.5^\circ, \text{Longitude } = 290.3^\circ \\
m_{EX} &= -\cos(\text{Longitude})\cos(\text{Latitude}) \\
m_{EY} &= -\sin(\text{Longitude})\cos(\text{Latitude}) \\
m_{EZ} &= -\sin(\text{Latitude})
\end{align*}
\]

(3.5)

After every simulation time step, the position of the right ascension of Greenwich (\(\alpha\)) was recalculated so that the current direction of the dipole moment in the inertial frame could be determined.

\[
\begin{align*}
m_{E1} &= m_{EX} \cos(\alpha) - m_{EY} \sin(\alpha) \\
m_{E2} &= m_{EX} \sin(\alpha) - m_{EY} \cos(\alpha) \\
m_{E3} &= m_{EZ}
\end{align*}
\]

(3.6)

Using this information, the magnetic field in the inertial frame at the position of the microsatellite was then calculated using the normalized position coordinates

\[
b_1 = \frac{H_0 R_E}{r_o^3} \left[3(m_{E1} \hat{r}_{e1} + m_{E2} \hat{r}_{e2} + m_{E3} \hat{r}_{e3}) \hat{r}_{ot} - m_{E1}\right]
\]

where \(R_E\) was the radius of Earth, \(\hat{r}_o\) was the orbital position in the inertial frame (Cartesian coordinates, normalized to a magnitude of 1.0), and \(H_0\) was the dipole strength at the surface of Earth. The value for \(H_0\) and the location of the magnetic North Pole were provided by Dynacon. These results were the outputs from the C UserCode block, which were then run through a block that rotated the magnetic field values to the body frame of the microsatellite. The elements of the rotation matrix came from the Attitude SuperBlock.

The Sun model was used to determine if the microsatellite was in sunlight or darkness. If it was in darkness, then the sun sensor would not be functional and would provide no
observations for ACS activities. The solar pointing frame was used to deal with the direction of the Sun.

The direction of the Sun was \( s_s = [1 \ 0 \ 0]^T \). To determine when the Sun was eclipsed, the inclination of the Earth with respect to the Sun had to be taken into account. The perifocal orbital coordinates generated by the orbital dynamics model must be rotated into a frame that included the proper Earth inclination \( (i_E) \) angle to take into account the current season: 23.5° for the Northern Summer Solstice (Jun. 21\textsuperscript{st}), -23.5° for the Northern Winter Solstice (Dec. 21\textsuperscript{st}), and 0° for the Vernal and Autumnal Equinoxes (Mar. 21\textsuperscript{st} and Sep. 21\textsuperscript{st}). The equation used was

\[
\cos(X) = -\sin(\theta)\sin(i)\sin(i_E) + \cos(\theta)\cos(\Omega)\cos(i_E) - \sin(\theta)\cos(i)\sin(\Omega)\cos(i_E) \tag{3.8}
\]

If the following condition was satisfied

\[
X < \gamma, \text{ where } \sin(\gamma) = \frac{R_E}{r_0} \tag{3.9}
\]

then the satellite was in eclipse. The variable \( r_0 \) was the radius of the orbit, \( \theta \) was the current true anomaly, and variable \( X \) was the angle from the Sun to the microsatellite in the \([\hat{r}_i \ \hat{r}_s]\) plane. This equation assumed that the orbit was circular \( (e=\omega=0) \). This equation also assumed that the orbit in the simulation was sun-synchronous. To simulate spacecraft in other orbits, this equation can be changed, though it will be more complex in terms of the number of variables involved. The functionality of Equations 3.8 and 3.9 was verified by comparing their results on the RT-Lab simulator to simulations run on Satellite Tool Kit (STK) [27].

The outputs from the Sun C UserCode blocks were \( s_s \) and the state of eclipse (1: no eclipse, 0: eclipse). The vector \( s_s \) was then run through a set of blocks that rotated it to the inertial frame and then to the body frame of the microsatellite. The outputs from this block were connected to a switch. The status of the switch was determined by the state of eclipse. If the microsatellite was in eclipse, the switch would output \([1 \ 0 \ 0]^T\) as a default value. This had the effect of removing the sun sensor observations from the attitude control algorithms.
The outputs from this SuperBlock were used by the sensors to produce observations for use by the ACS Processor SuperBlock. The magnetic field outputs were also used by the Magnetorquer SuperBlock so that the proper torques could be created. Finally, the state estimator must run its own software orbit propagator in order for it to work correctly. For the simulation, rather than having two separate orbit propagators running at the same time, it was decided to have one, with this one also sending its results to the state estimator with an appropriate error (~5%) introduced. This error represented the error that would occur between the orbit propagator of a real microsatellite and its actual position in orbit. The value 5% was chosen because it represented a relatively large error. Most on-board orbital propagators can maintain a 1% error or less with respect to the actual position of the spacecraft, and 1% was the error used in a spacecraft simulation performed for Dynacon [15]. If the sample ACS flight code could maintain a coarse pointing routine with this larger error, then it would help confirm its robustness.

**Attitude SuperBlock**

\[ \text{Master} \rightarrow \text{Environment} \rightarrow \text{Attitude} \]

The Attitude SuperBlock was used to model the attitude dynamics of the MOST simulation. The inputs into the Attitude SuperBlock included all the torques that the microsatellite can experience in the simulation. This includes the torques generated by the reaction wheels, the magnetorquers, the natural dipole moment of the microsatellite, and any disturbance torques introduced by the user via the Console interface. If any more disturbance torques, such as atmospheric drag and solar pressure, are added to the Orbit SuperBlock in the future, then they can easily be included in the total torque vector used by the attitude dynamics model. Note that the torques generated on the reaction wheels were subtracted from the total torque vector because the Attitude model required the torques they induced on the microsatellite, which of course are in the opposite direction of their own torques.

The attitude dynamics are simulated using the complete version of Euler's equations, including the gyro effects of the spinning reaction wheels that act like disturbance torques; the
reaction wheel speeds were the other three inputs into this SuperBlock. The attitude of the microsatellite was parameterized by the 3-2-1 Euler angle sequence $\theta = [\theta_1, \theta_2, \theta_3]^T$. Therefore, the matrix of rotation from the inertial frame to the body frame $\mathbf{\bar{z}}_b = [\bar{b}_1, \bar{b}_2, \bar{b}_3]$ is

$$
C_{b\infty}(\theta) = C_1(\theta_1)C_2(\theta_2)C_3(\theta_3)
$$

$$
= \begin{bmatrix}
    c(\theta_2)c(\theta_3) & c(\theta_2)s(\theta_3) & -s(\theta_2) \\
    s(\theta_1)s(\theta_2)c(\theta_3) - c(\theta_1)s(\theta_3) & c(\theta_1)c(\theta_2)c(\theta_3) + s(\theta_1)s(\theta_3) & s(\theta_1)c(\theta_2) \\
    c(\theta_1)s(\theta_2)c(\theta_3) + s(\theta_1)s(\theta_3) & c(\theta_1)s(\theta_2)s(\theta_3) - s(\theta_1)c(\theta_3) & c(\theta_1)c(\theta_2)
\end{bmatrix}
$$

where $C_1$, $C_2$, and $C_3$ are the matrices defining a rotation around the 1, 2, and 3 axes respectively. When the microsatellite was in its nominal pointing orientation, with the telescope pointing in the anti-solar direction (this assumes no star was being targeted), then $\theta_1 = \theta_2 = 0$ and $\theta_3 = \omega_3 t$. The angular velocity of the microsatellite is denoted by $\bar{\omega} = \bar{S}^T_w \omega$ where $\omega = [\omega_1, \omega_2, \omega_3]^T$. The attitude equations of motion are

$$
\dot{\omega}(t) = S^{-1}(\theta) \bar{\omega}
$$

$$
\bar{\omega}(t) = l^{-1}[\omega^* (l \omega + h_w(t)) + \mathbf{g}]
$$

where

$$
S^{-1}(\theta) = \begin{bmatrix}
1 & \sin \theta_1 & \cos \theta_1 \tan \theta_2 \\
0 & \cos \theta_1 & -\sin \theta_1 \\
0 & \sin \theta_1 \sec \theta_2 & \cos \theta_1 \sec \theta_2
\end{bmatrix}, \quad I = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
$$

The matrix $S^{-1}(\theta)$ is the kinematical relation linking $\theta$ with $\omega$, meaning that

$$
S = \begin{bmatrix}
I_1 & C_1 & I_2 \\
C_1 & C_1 C_2 & I_3
\end{bmatrix}
$$

It is important to observe that there would be a singularity whenever $\theta_2$ equaled exactly $\pm \pi/2$. Using a more complex system of Euler parameters to define the attitude orientation could have alleviated this problem. However, it was decided in the end not to change the method of parameterization. It was determined through experimentation that the chances of such a singularity occurring were rare when the spacecraft was tumbling (it only happened one time) and once the microsatellite was detumbled and running a coarse pointing attitude control routine, it would never approach this undesired angle of orientation.

The column vector $\mathbf{g}$ is the sum of all the torques, which when broken down becomes

$$
\mathbf{g} = \mathbf{g}_c + \mathbf{g}_m + \mathbf{g}_d
$$
where $g_c$ are the control torques induced by the reaction wheels, $g_m$ is the total magnetic torque induced by both the magnetorquers ($g_{mc}$) and the natural dipole moment ($g_{md}$), and $g_u$ was the user-controlled disturbance torque. The column vector $h_w$ is the angular momentum of the three reaction wheels, which are three of the inputs into this SuperBlock. The term $\omega^t h_w(t)$ is a time-varying disturbance torque introduced by the gyro dynamics of the spinning reaction wheels.

The attitude dynamics model was written using a BlockScript block. BlockScript is the native language of SystemBuild and allows you to do complex calculations in your model. Though not as flexible as C, it was preferable to use BlockScript when it came to writing simple calculating routines that would never be used in the future as flight code. In BlockScript, the variables $U_i$ ($i = 1, 2, ..., m$) are the input values, $Y_j$ ($j = 1, 2, ..., n$) are the output values, and $X_k$ ($k = 1, 2, ..., p$) are the state values. The initial values of the states, in this case the Euler angles ($X_1, X_2, X_3$) and the rates ($X_3, X_4, X_5$), were set using the Real Parameters list of the BlockScript block, which was accessed using the Block Parameters button.

The output from the Attitude SuperBlock included not only $\Theta(t)$ and $\Omega(t)$, but also the nine components of the matrix $C_{bi}$. These components are used by the Orbit SuperBlock to determine the body frame values of the magnetic field of the Earth and the direction of the Sun for use by the magnetometer, sun sensor, and magnetorquer emulations.

Satellite SuperBlock

$Master \rightarrow Satellite$

The emulation of the microsatellite contained three SuperBlocks: the Sensor model, the Actuator model, and the ACS DSP model. The inputs into the Sensor SuperBlock came from the Environment SuperBlock with all the outputs going into the ACS DSP model. The Actuator SuperBlock received all its inputs/commands from the ACS SuperBlock, along with the magnetic field in the body frame (for use by the Magnetorquer emulation). The Actuator outputs were fed back to the Environment SuperBlock as well as the ACS SuperBlock for use by the state estimator model and some of the outputs were sent to the user console display.
All of the sample ACS flight code written was placed in the ACS SuperBlock, with one exception (see the Reaction Wheel SuperBlock). All of this code, with some modification could be used, in principle, on the actual ACS DSP.

Sensor SuperBlock

Master → Satellite → Sensor

The Sensor SuperBlock contained three other SuperBlocks: Magnetometers, Sun Sensor, and Rate Sensors. The contents of all three SuperBlocks were the same, three inputs each coming from the environment model and three outputs each with the same value as the input along with some added sensor noise. For the three-axis magnetometer emulation, the three inputs are the satellite body frame values of the magnetic field of the Earth. For the two-axis sun sensor emulation, the three inputs are the direction vector of the Sun with respect to the x-axis face of MOST, with the aperture of the CCD camera on the negative x-axis face of microsatellite. When MOST is in its nominal anti-solar pointing configuration, the sun direction vector would be

\[ [1 \ 0 \ 0]^T = s_t \]

Again, this assumed that MOST was in a dawn-dusk sun-synchronous orbit. If the microsatellite was in eclipse, the inputs into the sun sensor emulation would default to \([1 \ 0 \ 0]^T\). Doing this was the closest way to simulate the removal of the sun sensor from any state estimation algorithms. Finally, the three inputs for the rate sensor emulation were the rates of rotation around each body axis.

It was assumed that the magnetometer and rate sensors are aligned along the principal axis frame of the microsatellite. The addition of rotation matrix blocks into this SuperBlock would correct any discrepancy if the sensors were placed in a different orientation in the spacecraft. However, at this time, the exact orientation of each sensor was not known. Therefore, it was decided to use the principal axis for now.
It is apparent that these sensor emulations are very simple and much of their design is “throw-away” work that will have to be modified before the ACS processor hardware can be connected as hardware-in-the-loop. However, the sensor hardware that was to be used on MOST had yet to be defined, so it was impossible to convert the outputs of every sensor into the proper voltages and currents. Chapter 6 details the changes done to these emulations when the “throw-away” work was removed.

The sensor noise, defined as the column vector \(v_i(t)\) with \(i = 1, 2, 3\) for the three sensors, was additive with discrete-time covariance

\[
R^{(i)} = \varepsilon \{v^i(t_k) v^T(t_k)\} = \tau_i 1 \delta_{tt} \quad [15]
\]

where \(t_k\) represents a set of equally spaced sample times (the simulation step period, which is 0.1 sec.) and \(\varepsilon\{\cdot\}\) is the expectation operator. From this, the value of each component of \(v_i(t)\) was generated in the simulation using the following:

\[
v_j^{(i)} = \sqrt{2\tau_i} \left[ \sigma_{i,j} - \frac{1}{2} \right] \quad [15]
\]

where \(j = 1, 2, 3\) for each component and \(\sigma_{i,j}\) is a random variable between 0 and 1 with uniform distribution. The random number and noise generation was done using SystemBuild’s BlockScript language, as was done for the attitude calculations in the Attitude SuperBlock. The values of \(\tau_i\) were \(2 \times 10^{-7}\), 0.001333, and 0.01167 for the magnetometer, rate sensors, and sun sensor respectively.

The rate sensor hardware for MOST was included with the Dynacon reaction wheel electronics. When it comes to determining the output of the rate sensors, the results from the serial data stream of the reaction wheel must be decoded (see Hardware Reaction Wheel SuperBlock). For this simulation model, it was decided to place the rate sensor emulation separate from the reaction wheel emulation. The reason for this was because if you wanted to temporarily replace the reaction wheel emulation with the actual hardware to help refine the emulation design (as was done for this simulation), the rate sensor would be useless unless a one-axis air-bearing table was also available. In this case, one was not available and the reaction wheel hardware sat on a table, rendering the rate sensor useless and necessitating the use of an
emulated rate sensor. When it came time to modify the simulation for use in its secondary purpose as operations support, the rate sensor was incorporated into the reaction wheel emulation.

Actuator SuperBlock

\[ \text{Master} \rightarrow \text{Satellite} \rightarrow \text{Actuator} \]

This SuperBlock contains the emulations of both actuator systems used by MOST. The inputs into this SuperBlock included the serial data streams commanding all three reaction wheels (each stream contained 9 bytes), the dipole moment commands for the magnetorquers, and the magnetic field of the Earth expressed in the body frame. The outputs from the SuperBlock were the response serial data streams from the three reaction wheels (each stream again contained 9 bytes), and the torques generated by all six actuators. It was assumed that all the actuators were aligned along the principal axis of the microsatellite. Again, it was not known at this time the exact orientation of the actuators in the spacecraft, so that was why this assumption was made. However, as long as the actuators are placed in a body axis centered on the centroid of the spacecraft, then the addition of a rotation matrix block for each actuator would correct any discrepancy.

Magnetorquer SuperBlock

\[ \text{Master} \rightarrow \text{Satellite} \rightarrow \text{Actuator} \rightarrow \text{Magnetorquer} \]

Though this emulation contains many blocks, including 3 SuperBlocks, its design was very simple. The microsatellite had magnetorquer coils aligned along three axes of the spacecraft, represented in this SuperBlock by the blocks Magnetorquer1\_axis, Magnetorquer2\_axis, and Magnetorquer3\_axis. Each magnetorquer received a simulated electrical current as command by either the user or the attitude control algorithm. The induced magnetic dipole moment could be both positive and negative depending on the polarity of the applied current and each coil had a maximum/minimum limit on the dipole moment it could induce (see Table 3.1, in Section 3.2).
The total dipole moment vector due to the magnetorquers was then added to the natural
dipole moment of the microsatellite, which at the time was estimated to be 0.1 Am² along each
principal axis for MOST (this estimation came from Dynacon). The total magnetic torque
induced by this dipole moment was

\[ g_m = m^\top b \]  \hspace{1cm} (3.18)

where \( m = [m_1 \ m_2 \ m_3] \) is the total magnetic moment of the microsatellite and \( b \) is the magnetic
field of Earth expressed in the body frame of the spacecraft. The magnetic torque vector is the
only output from this emulation for use by the Attitude SuperBlock.

**Reaction Wheel SuperBlock**

\textit{Master \rightarrow Satellite \rightarrow Actuator \rightarrow Reaction Wheel}

Three reaction wheel emulations were contained in this SuperBlock. During the
development of the simulation, a hardware reaction wheel became available. It was connected to
the simulator as hardware-in-the-loop and interacted with the simulation. This was done to aid in
the development of the software emulation of this actuator. Therefore, two SuperBlocks were
created for the reaction wheel, one for interacting with the hardware and the other a software
emulation.

The other three blocks found in this SuperBlock were C UserCode blocks containing
sample flight code used to estimate the torques induced by each reaction wheel based on the
outgoing serial data generated by the actuators, which contained the wheel speeds. These torque
estimations were used in the Environment SuperBlock and by the state estimation algorithm.
The functionality of the code was quite simple; after decoding the rotation speed of the reaction
wheel from two bytes of the 9-byte data stream, it estimated the torque of each reaction wheel using

\[ g_{ci} = \frac{I_{wi}(\omega_{ci} - \omega_{ai})}{T_s}, i = 1, 2, 3 \]  \hspace{1cm} (3.19)

where \( I_{wi} \) is the moment of inertia of each reaction wheel, \( \omega_{ci} \) is the current rotational speed of
each reaction wheel (rad/sec.), \( \omega_{ai} \) is the rotational speed of each reaction wheel from the
previous time step of the simulation, and \( T_s \) is the time step length of the simulation (sec.).
a short time step of 0.1 sec., this method of estimating the reaction wheel torques was found to work well for performing accurate state estimation.

It should be noted that this is the only instance where sample ACS code was not located within the ACS Processor SuperBlock. This was done because simulation also required the reaction wheel torques to properly execute the attitude dynamics model in the Environment SuperBlock. Placing the torque C UserCode blocks in the ACS SuperBlock caused some circular algebraic logic errors in the simulation model. One solution was to place the blocks in the Reaction Wheel SuperBlock. It was later learned that skewing the ACS SuperBlock back one time step (0.1 sec.) would have also solved the problem, however it was decided to leave these UserCode blocks in this location.

**Hardware Reaction Wheel SuperBlock**

*Master → Satellite → Actuator → Reaction Wheel → Hardware Reaction Wheel*

This SuperBlock consisted of only three blocks, both custom-made for RT-Lab simulations. The first block initialized the hardware drivers that prepared the simulation for asynchronous serial communication by initializing the IP501 RS-422 serial card, to which the reaction wheel hardware was connected. The IP501 had four serial ports, and the card itself sat in one of four slots on the ATC Greenspring motherboard (see Appendix A). These variables had to be defined by the user in the parameters of these IP501 Asynchronous blocks so the serial packet could be sent properly. Two IP501 Asynchronous blocks were required because two different serial ports had to be defined: one for sending and one for receiving. Because this was a beta version of the IP501 code, it was missing many features, including the ability to transmit and receive on the same IP501 port. This problem has since been fixed for the latest version of the simulation (see Chapter 6).

The second block received the 9-byte serial packet created by the ACS SuperBlock for commanding the reaction wheel. The block then sent the command to the IP501. The IP501 could send data at numerous baud rates. At the time of its use, the IP501 Send block was still being beta tested, so some of its features were not very user-friendly. For example, in order to
change the baud rate, it required going into the C source code of the IP501 Send block and manually changing the variable that defined the baud rate. These changes were not too difficult to make; it only required learning where to find the code and where in the code the change had to be made. For the hardware connection to the reaction wheel, Port 0 (A as defined by the IP501 documentation) was used for sending the serial packet, the IP501 was sitting in Slot 1 (B as defined by the ATC documentation), and the data transmission was done at 19.2 kBaud, 8 bits, 1 stop bit, and no parity. See Figure 3.4 for a diagram of the wiring connection.

The third block received the response 9-byte serial packet from the reaction wheel and sent it to both the ACS SuperBlock and the torque C Usercode Blocks (see Reaction Wheel SuperBlock). Again, all of the variables defining the physical location of the IP501 and the transmission rate had to be defined. All of the settings were the same as the IP501 Send block except for the port location. The IP501 Receive block was set to receive the serial packet from Port 1 (B as defined by the IP501 documentation). One note on the reaction wheel: though capable of asynchronous communications, the serial communication line was only half-duplex. While sending a reply packet, it may ignore any incoming command packet. See Figure 3.4 for a diagram of the wiring connection.

As has been previously mentioned, all commands given to the reaction wheel were in the form of a 9-byte serial packet. After receiving a command, the reaction wheel would execute it and reply with another 9-byte serial packet. Figure 3.5 gives a breakdown on the various bytes that make up both packets. The first and last bytes, "<" and " >" respectively, are markers used by the reaction wheel software to help define a valid packet: one that contains nine bytes with these two markers at the beginning and end. After receiving the "<" byte, the serial buffer of the reaction wheel resets and accepts eight more bytes. If the " >" byte is not found as the last byte, the packet is discarded until the next "<" byte is received.

The mode byte is used in the command packet to place the reaction wheel into one of its numerous command modes. The value of the mode byte in the response packet is the current command mode of the reaction wheel. The modes that were used in the simulation were:
where...

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>BOS</td>
<td>LBL</td>
<td>m</td>
<td>i</td>
<td>da</td>
<td>db</td>
<td>dc</td>
<td>dd</td>
</tr>
</tbody>
</table>

- **BOS** = @<@ Start of sequence delimiter, ASCII code 60
- **LBL** Label byte, user defined
- **m** Mode/command identifier
- **i** Information identifier, use is mode dependent
- **da..dd** Data bytes, use is mode dependent
- **EOS** = A>@ End of sequence delimiter, ASCII code 62

Null/Query: Can be called at any time to access reaction wheel telemetry. It does not change the current mode of the reaction wheel (the data bytes are ignored).

**Built-in Test**: This mode is only used to turn on the rate sensor when the simulation is initialized.

**Disabled**: This is the starting mode of the reaction wheel when it is powered up. Most of the wheel circuitry is turned off (the data bytes are ignored).

**Open-Loop Voltage**: The reaction wheel motor will spin up to a certain speed depending on what voltage it is commanded to reach.

**Speed**: This is a closed-loop feedback control mode. The wheel will spin up to a certain speed as commanded by the user.
Torque: This is a closed-loop control mode. The wheel will spin up at a certain torque as commanded by the user.

Note: the Dynacon reaction wheel has its own built-in proportional-integral (PI) microcontroller electronics to run the Speed and Torque modes.

The label byte can be any value as defined by the user. Its purpose is to correlate a response packet with its command packet; the user gives a specific label byte to a specific command and the response packet for that command will have the same label.

The four data bytes are used in the command packet to command the reaction wheel to a certain open- or closed-loop speed or torque (as appropriate for the current mode of the wheel). All the data must be low byte first and 2’s-complement form is used to store negative values. A scaling factor is applied to the data to allow for decent float-to-integer conversion. In the reply packet, the four data bytes are used to return wheel telemetry. The first two bytes always return the estimated wheel speed of the reaction wheel. This estimate is generated by the internal processor of the reaction wheel and is reliably accurate. The telemetry value returned by the last two data bytes depends on the value of the information identifier byte in the command packet. The information identifier bytes used in the simulation were: current mode (1), closed-loop wheel speed error (6), motor supply voltage (41), rate sensor (15), and wheel torque (60). The reaction wheel, at the time, could output only one of these telemetry values per response packet. Therefore, it was decided to create a separate torque estimation routine in the Reaction Wheel SuperBlock so that the rate sensor telemetry could be made available without any torque telemetry, which was required by the Environment SuperBlock and the state estimation algorithm. Again, all the data returned must be scaled to get the actual floating point value. The wheel torque telemetry has two separate scaling factors depending on the range of the telemetry (low or high). The information identifier byte in the return packet informs the user on which range was used so that the proper scaling factor can be applied. Scaling the rate sensor data is not just a simple matter of dividing it by a constant. The following formula must be applied

\[
\text{Rate sensor output (rad/sec.)} = ((\text{data value} - 512.0) / 16000.0) - 1.895 \times 0.087266
\]
where 512.0 and 16000.0 are scaling factors, 1.895 is the bias from the zero point of the sensors, and 0.087266 converts the resulting voltage into rad/sec. (5.0 is used instead of 0.087266 to convert the voltage into deg/sec.).

The first test done using the reaction wheel was to determine its moment of inertia. The reaction wheel was commanded to a torque of 0.001 Nm for 600 seconds. The wheel speed was measured every 0.1 seconds and the results are shown in Figure 3.6. With a slope of 0.5935 and an intercept very close to zero, the moment of inertia was calculated to be

\[ I_w = \frac{g_w}{\omega} = \frac{0.001 \text{ N m}}{0.5935 \cdot 10^{-1} \text{ rad/s}^2} = 0.0001685 \text{ kg m}^2 / \text{ rad} \]

This value was very close to the typical moment of inertia for a Dynacon reaction wheel (around 0.000165 kg m²/rad) and thus was used in the software emulation model of the wheel.

The next test performed was to determine the relationship between the commanded voltage and the reaction wheel speed in the open-loop voltage mode. Figure 3.7 shows the results of this test; the slope value of 43.385 rad/V-sec was included into the software emulation of the reaction wheel so that this mode could be simulated properly.

When the PI controller for the software emulation of the wheel was designed (see Software Reaction Wheel SuperBlock), the results were compared to the hardware wheel to confirm that the time constants and overshoot values for the closed-loop modes were similar. Finally, the hardware wheel was periodically inserted into the simulation for various runs to learn as much as possible about its behavior. For example, it was found that the closed-loop controller performed poorly in the hardware reaction wheel whenever the magnitude of the wheel speed was less than 5 rad/sec. (see the poor position control example in Figure 3.8). Therefore, the wheel speeds should never be kept near these values once they have been removed from the idle mode. In open-loop voltage mode, the hardware reaction wheel speed was 0.0 rad/sec if the magnitude of the commanding voltage was less than or equal to 0.62 V (see Figure 3.7).
Figure 3.6: Reaction Wheel Moment of Inertia Calculation

Wheel Acceleration (RW #102) (torque=0.001 N-m)

\[ y = 0.5935x - 1.566 \]

Time (ds)

Figure 3.7: Voltage to Wheel Speed Relationship (Open-Loop Voltage Mode)

Voltage vs. Wheel Speed of RW #102 (Open-Loop Voltage Mode)

\[ y = 43.385x + 0.0664 \]

Voltage (V)

Figure 3.8: Poor Performance of HW Reaction Wheel at <10 rad/sec. (Axis 1 – black line – was controlled using the hardware reaction wheel. The other axes had software emulations of the wheel)
Two C UserCode blocks were required in the design of the software emulated reaction wheel. The first block received the 9-byte packet and determined the commanded mode. If the commanded mode was not Null/Query, it changed the inputs going into the wheel emulation so that the wheel was placed into its new control configuration. The second block created the response 9-byte packet, including the wheel speed telemetry and the data that was requested by the command packet. Though the hardware reaction wheel could supply many different telemetry types, the software emulation only supplied the following telemetry: current mode, closed-loop wheel speed error, motor supply voltage, rate sensor, and wheel torque. This telemetry was useful in the design of the sample ACS flight code, while the other available telemetry (e.g., wheel temperature, internal pressure) were not needed and would be difficult to emulate in any useful way.

Once the command packet was processed by the C code, the currently desired wheel speed was sent to the Wheel Plant SuperBlock and the Wheel Plant/PI Controller SuperBlock connected to a switch. The status of the switch was determined by the control mode of the wheel. If the wheel was in the open-loop voltage mode, the Wheel Plant SuperBlock was used, otherwise the closed loop Wheel Plant/PI Controller SuperBlock was used. The results from the switch were then passed though a saturation block that limited the speed of the reaction wheel to ±500 rad/sec.

Ignoring the transient response due to coil inductance, the reaction wheel dynamics can be modeled using the following equation [25]

\[
d_1 + d_2 \omega_w(t) + d_3 \dot{\omega}_w(t) = v_w(t)
\]

\[
d_1 = \frac{R_2}{K_i} \tau_f \quad d_2 = K_n + \frac{R_2}{K_i} B \quad d_3 = \frac{R_2}{K_i} I_w
\]

where

- \(\omega_w\) wheel speed
- \(I_w\) wheel moment of inertia
- \(K_i\) effective motor torque constant
The values of $\tau_f$ and $B$ are very small for the reaction wheel. Therefore, they were set to zero to simplify the software emulation design. This resulted in $d_1 \equiv 0$ and $d_2 \equiv K_b$. Using a Laplace transform, the voltage to wheel speed transfer function was determined

$$W(s) = \frac{1}{d_1 + d_3 s} V(s)$$

Using the results shown in Figure 3.7 (all the data were collected when $\dot{\omega} = 0$), $d_2$ was calculated to be 0.02305 V-sec/rad. The typical armature resistance for the Dynacon reaction wheel is 2 $\Omega$. Since $K_i = K_b$, $d_3$ was calculated to be 0.01458 V-sec$^2$/rad. This model was placed in the Wheel Plant SuperBlock.

This model was also placed in the Wheel Plant/PI Controller connected to a PI controller in a closed loop. The resulting transfer function was

$$\frac{K_p s + K_l}{d_3 s^2 + (d_2 + K_p) s + K_1}$$

where $K_p$ and $K_l$ are the proportional and integral gains, respectively, and $d_2$ is a disturbance term.

$$1 + \left( \frac{K_p s + K_l}{d_3 s^2 + (d_2 + K_p) s + K_1} \right)$$

$$= \frac{K_p s + K_l}{d_3 s^2 + (d_2 + K_p) s + K_1}$$

$$= \frac{K_p s + K_l}{d_3 s^2 + (d_2 + K_p) s + K_1}$$

where $K_p$ and $K_l$ are the proportional and integral gains, respectively, and $d_2$ is a disturbance term.
The PI controller and feedback made the plant into a Type I model (ie. one free integrator), which meant that there was zero steady-state error for a step command. When using the closed loop Speed mode with the hardware reaction wheel, it was observed that there was some overshoot and the reaction wheel reached the desired speed quickly. For the software emulation, it was determined that a time constant ($T_c$) of 5 sec. and a damping ratio ($\zeta_c$) of 0.75 would be a close approximation to the hardware reaction wheel interfaced previously to the simulator. The proportional and integrator constants ($K_p$ and $K_i$), which were not known for the hardware reaction wheel, were calculated to be

$$\omega_n = \frac{2\pi}{T_c}$$

$$\frac{(d_2 + K_p)}{d_3} = 2\zeta_c \omega_n \quad \vdash K_p = 0.1884 \text{ V - sec}$$

$$\frac{K_i}{d_3} = \omega_n^2 \quad \vdash K_i = 0.1256 \text{ V - rad}$$

When using these values, the closed-loop operation of the software was found to be similar to the closed-loop operation of the hardware reaction wheel. The feedback error was one of the outputs for the Wheel Plant/PI Controller SuperBlock so that it could be included as reaction wheel telemetry.

The hardware reaction wheel on which the DC motor plant and PI controller were based was an older, engineering model of the wheel. These values will be different for the reaction wheels used on MOST. Once the characteristics of these wheels are known, the simulator can be updated with new values for the motor plant and controller.

**ACS Processor SuperBlock**

$$\text{Master} \to \text{Satellite} \to \text{ACS Processor}$$

There were many blocks used in this SuperBlock. All of the C UserCode blocks used here, along with the torque C UserCode blocks in the Reaction Wheel SuperBlock, contained all of the sample flight code that could, in principle, be used in some form on the actual ACS
The C code used for the orbital environment models can also be used as the on-board orbit propagator with some modifications.

The three gain blocks on the left-hand side of the SuperBlock were used to condition some data before being used by the ACS code. The first one was used to add in some error to the sensor observations (which already include noise error) if the user so desired. This was not done for the current simulation. The second gain block inverted the direction of the torques coming from the reaction wheel emulations. This was done because the state estimator required the torques they induced on the microsatellite, which of course were in the opposite direction of their own torques. Finally, the third gain block was used to introduce some error into the true anomaly and magnetic field values (body frame) before they were used by the state estimator. This was done to simulate the ACS Processor having its own on-board orbit propagator that was not exactly synchronized with its actual orbit.

The three RW Input Code blocks each ran the same C flight code function. This function generated the 9-byte packet that commanded the reaction wheel emulations, with three copies for three reaction wheels. The inputs into each C UserCode block were the desired reaction wheel mode and the command (if any) for that mode. All of the inputs came from the ACS Flight Code block. The three RW Output Code blocks also ran the same function. This function received the 9-byte response packet from the reaction wheel emulations and processed the packet to determine the spinning speed of the reaction wheel as well as the telemetry data that was requested. As was stated previously, the final version of the simulation had fixed this requested telemetry to be the rate sensor (torque telemetry was separately calculated). These two results were the outputs from the C UserCode block. The wheel speed was used by the Attitude SuperBlock so that the gyric effect of the spinning wheels could be included in the attitude dynamics of the microsatellite. Since the reaction wheel speeds were actual observations available to the ACS processor, they were also used by the state estimator so that its attitude equations would closely model the "real" dynamics of the microsatellite.

The ACS Flight Code C UserCode block contained the code for controlling every actuator on the microsatellite (magnetorquers and reaction wheels). This C UserCode block also
processed all of the observations coming from the sensors. The actuator commands and processed sensors readings were the only outputs from this block. Every time step, using the observations from the magnetometer, this code also calculated the rate of change with respect to time of the magnetic field along each body axis of the microsatellite using

$$\vec{b}_o^{(n)} = \frac{1}{T_s} (\vec{b}_o^{(n)} - \vec{b}_o^{(n-1)})$$  \hspace{1cm} (3.24)

where $\vec{b}_o = [b_{o1}, b_{o2}, b_{o3}]^T$ was the magnetic field observed by the magnetometer, $n$ was the current time step of the simulation, and $T_s$ was the time step length of the simulation (sec.). With a short time step of 0.1 sec., this method is accurate for the simulation. The vector $\vec{b}_o$ was used by the ACS detumbling routine.

In Section 3.2, it is shown that the user could place the simulation in four modes. The sections of code used in the ACS Flight Code block depend on the mode selected in the Console display. If the user places the simulation in *HK Override* mode, the code would control the actuators depending on the inputs given by the user in the Console. The magnetorquers could be switched off or on to the maximum magnetic dipole moment they could create (in either the positive or negative direction along its axis of orientation). The user could also slew the microsatellite around on each axis using the reaction wheels. To do this, the ACS Flight Code block placed the reaction wheel into its closed-loop Speed mode and spun the wheel up to the user-specified speed. The duration of the slew, in seconds, was

$$T_{\text{slew}} = \frac{\beta}{\left(\frac{I_w}{I_j}\right)\omega_w}$$  \hspace{1cm} (3.25)

where $\beta$ was the user-specified slew angle (in radians) and $I_j$ was the moment of inertia of the microsatellite along the slew axis.

If the simulation was in *Detumble* mode, the ACS Flight Code block used the magnetorquers to reduce the current tumbling speed of the microsatellite to 0.25°/sec. along each body axis. No matter how great the initial tumbling speed, the code would always be successful,
though it might take several orbits before the microsatellite was detumbled. Details on the
detumbling algorithm, along with some simulation test results, can be found in Chapter 5.

If the simulation was in Coarse Pointing mode (with or without Reaction Wheel
Desaturation), the ACS Flight Code block used the reaction wheels to point the microsatellite so
that its body frame matched the solar pointing frame. This placed the aperture of the telescope in
the direction of the anti-solar CVZ (see Figure 1.8). Though not modeled in the simulation, once
pointing in this direction, the star-tracker could then be used to point MOST at a specific star for
study. The coarse pointing algorithm could point the microsatellite within less than one arc-
minute of the desired direction along each axis. The rates of the microsatellite could also be
reduced to less than 0.05 deg/sec. For coarse pointing, the code placed the reaction wheels into
their closed-loop Torque mode. The torque commands given to the wheels were dependent on
the results of the satellite state estimator, the final block found in the ACS Processor SuperBlock.
The state estimator, using the observations from the sensors, the on-board orbit propagator, the
estimated reaction wheel speeds and torques, and the commands given to the magnetorquers,
attempted to determine the current angle (with respect to the inertial frame) and rate states of the
microsatellite. Using a Kalman Filter, the state estimator could quickly converge to the actual
microsatellite states, even with the presence of non-modeled disturbance torques, as long as the
initial states of the spacecraft were within the following limitations: ±50° around each axis with
respect to the inertial frame with a ±1°/sec. spin around each axis; or ±80° around each axis with
respect to the inertial frame with a ±0.25°/sec. spin around each axis. This was sufficient for
MOST because at and beyond these angles, the sun sensor would no longer be pointing at the
Sun, and without that sensor, the Kalman Filter and state estimator would no longer work
properly. More details about coarse pointing algorithm and the state estimation algorithm, along
with some simulation test results, can be found in Chapter 5.

If the simulation was in Coarse Pointing with Reaction Wheel Desaturation mode, the
ACS flight code block ran a routine using the magnetorquers to dump momentum from the
reaction wheels while they were being used to point the microsatellite. This momentum
dumping reduced the speed of the reaction wheels and prevented them from every approaching
their saturation speed, which was modeled to be 1000 rad/sec. for the software emulated wheels.
The hardware reaction wheel did not operate properly when its speed was lower than 5 rad/sec. (see Figure 3.8). Therefore, the desaturation routine would reduce the wheel speeds to 50 rad/sec. rather than to 0 rad/sec. It would be easy to modify the flight code to change the desaturated speed if so desired. Again, details about the momentum desaturation algorithm along with some test results can be found in Chapter 5.

3.1.2. Console SuperBlock

An OpComm block was in this SuperBlock. This block, along with its complement in the Master SuperBlock was used to define the boundaries of the Master subsection of the simulation, the subsection which will run on the QNX Target node of the simulator. All inputs passed through this block unchanged.

The four-button block was to allow the user to place the simulated microsatellite in one of four modes: HK Override, Detumble, Coarse Pointing, and Coarse Pointing with Reaction Wheel Desaturation. HK Override allowed the user to manually slew the microsatellite using the controls in the three Primary Axis SuperBlocks as well as manually test each magnetorquer (see next section). The other three modes placed the microsatellite into its own automated ACS routine allowing the spacecraft to control the actuators as necessary. The user had no direct control over the actuators in these modes.

Most of the outputs from the Console SuperBlock were fed back to the Master SuperBlock. However some of the outputs were set as outputs from the top level SuperBlock (see arrows on right-hand side of Console block in Figure 2.3). Once a simulation run was completed, this data would be available in XMath as a variable that the user could analyze and plot.
Primary Axis SuperBlock

Console → Primary Axis 1,2,3

All three Primary Axis SuperBlocks displayed the results coming from all three sensors. General orbital environment data, data that was not directly observed by the sensors, was also displayed. This data included the attitude Euler angles with respect to the inertial frame, the orbital position vector of the microsatellite in the inertial frame normalized to a magnitude of one, and the total magnetic torques that the microsatellite was producing (including those produced by the natural dipole moment of the spacecraft). Each Primary Axis SuperBlock also displayed the telemetry data coming from its respective reaction wheel: wheel speed and the user selected data.

If the simulation was in HK Override mode (see Console SuperBlock Section), the two slides and button allowed the user to slew the microsatellite around any of its three primary axes. The slides set the size, in degrees, of the slew, as well as the maximum speed of the reaction wheel. The “Zero Speed” button was used to quickly stop the reaction wheel in case of any problems.

Disturbance Torque SuperBlock

Console → Disturbance Torque

This SuperBlock allowed the user to introduce disturbance torques along each body axis into the simulation. This allowed the testing of the capabilities of the ACS flight code and its ability to handle non-modeled torques.

3.2. Simulation Execution

The important model parameters used in the simulation are listed in Table 3.1. These parameters were kept constant throughout the testing of the simulation. Every time a new ACS sub-system was emulated, the simulation was executed so that it could be debugged and have its functionality tested. Once all the sub-system emulations were finished and the sample flight
code was being written, the simulation was executed to debug and test the code. Finally, the simulation was used to test the completed sample ACS flight code over long time durations.

For the long duration tests, the time vector used in the SystemBuild Simulation window (Figure 2.3) was either [0:0.1:12000]' (0 to 12000 sec.) or [0:0.1:30000]' (0 to 30000 sec.). In SystemBuild, the first value was the start time, the second value was the time step, and the third value was the finish time. With an orbital period of around 6000 seconds, such time durations would account for 2 to 5 orbits. For the sub-system tests, much shorter durations of 100 to 1000 seconds were used. The Variable Kutta-Merson integration algorithm was used for all simulation runs.

<table>
<thead>
<tr>
<th>Table 3.1: Simulator Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Orbit</strong></td>
</tr>
<tr>
<td><strong>Principal Moments of Inertia [kg -m²]</strong></td>
</tr>
<tr>
<td><strong>Reaction Wheel Moment of Inertia [kg -m²]</strong></td>
</tr>
<tr>
<td><strong>Max. Magnetorquer Magnetic Moments [A-m²]</strong></td>
</tr>
<tr>
<td><strong>Sensor Noise (σ: random number between 0 and 1 with uniform distribution)</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>On-board Orbit Propagator for ACS Estimator</strong></td>
</tr>
<tr>
<td><strong>Simulation Step Period</strong></td>
</tr>
<tr>
<td><strong>Actuator Alignment</strong></td>
</tr>
</tbody>
</table>

All the RT-Lab simulations were run in “Software Synchronized” real-time mode (see Figure 2.2). To help speed up the simulation run, the “Time Factor” was set anywhere from 0.5 to 0.1, which reduced the run time of the simulation from one-half to one-tenth of the regular run time. However, when the reaction wheel was interfaced with the simulation as hardware-in-the-
loop to help design the software wheel emulation, the “Time Factor” had to be set to 1.0 so that
the wheel could respond properly to the commands it received from the simulation.

3.3. Simulation Summary

The emulation strategies described in the Section 2.2 were employed and sped up the
development of sample flight code. Table 3.2 lists by system the number of SystemBuild
mathematical blocks and lines of C code were used to model each system. The C code listed for
the ACS model includes the sample flight code that was written. In summary, the sample flight
code written covered the following functionality:

- Serial Communication With Reaction Wheels
- Actuator Torque Estimation
- State Estimator/Kalman Filter
- Detumbling Control Law
- Coarse Pointing Control Law
- Momentum Desaturation Control Law

The majority of systems were quickly and easily emulated using only SystemBuild
blocks, which saved much time. It took only three months to develop all of the flight code and
environment code, including the time required to develop the entire model and test the
functionality of the flight code. Five-sixths of that time was spent writing code while the rest of
that time was spent placing and linking the simulation blocks. Table 3.2 is ordered from top to
bottom by the complexity of the emulation required to model each system. Low complexity
subsystem emulations were those which required only built-in SystemBuild mathematical blocks
to create (no C UserCode blocks were required). All of the sensor emulations were of the same
complexity, while the magnetorquer emulations required more SystemBuild blocks, making
them slightly more complex. The reaction wheels, environment model, and ACS processor
subsystems required much C code in order to simulate. Thus, their emulations were more
complex. Complexity reflected how long it took to create the emulation and the difficulty in
creating the emulation. The sensors were the quickest and easiest to emulate, while the ACS processor model was the most difficult and required more time to create.

The quick development time was also made possible because testing and debugging the simulation and sample flight code was done on a block-model simulator system. The graphical aspect of the system made it simpler to spot errors and the software Console interface to the model made it easy to control the simulation and create different scenarios to test the ACS code and the fidelity of the actuator and sensor emulations.

Approximately one-sixth of the three month development time was spent working with SystemBuild mathematical blocks while the rest of the time was spent writing, debugging, and testing C code. Based on that timeline of three months, the approximate work time required to emulate each system is also listed in Table 3.2.

<table>
<thead>
<tr>
<th>System</th>
<th>No. Blocks Required</th>
<th>Lines of Code Required</th>
<th>Approx. Work (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnetometer Model</td>
<td>2</td>
<td>0</td>
<td>0.7</td>
</tr>
<tr>
<td>Sun Sensor Model</td>
<td>2</td>
<td>0</td>
<td>0.7</td>
</tr>
<tr>
<td>Rate Sensor Models (x3)</td>
<td>6</td>
<td>0</td>
<td>2.2</td>
</tr>
<tr>
<td>Magnetorquer Models (x3)</td>
<td>14</td>
<td>0</td>
<td>5.1</td>
</tr>
<tr>
<td>Reaction Wheel Models (x3)</td>
<td>6</td>
<td>155</td>
<td>11.1</td>
</tr>
<tr>
<td>Environment Model</td>
<td>8</td>
<td>360</td>
<td>23.6</td>
</tr>
<tr>
<td>ACS Model</td>
<td>3</td>
<td>790</td>
<td>46.5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>41</strong></td>
<td><strong>1305</strong></td>
<td><strong>90.0</strong></td>
</tr>
</tbody>
</table>
4. MOST ACS Flight Code Simulation Analysis

4.1. Model Development Methodology

Using the simulator system made it possible to develop important ACS code in three months, even without the presence of the ACS hardware. Assuming a typical processor development time of around 8 to 10 months (based on the MOST program), this allows concurrent hardware and software and hardware development, which shortens the amount of time that will be spent developing software after the ACS processor is built.

Once the ACS processor hardware is available, it would be beneficial if the simulator could still be used to work on the microsatellite, with the ACS processor connected as hardware-in-the-loop, as an operations support tool (see Chapter 6). From the experience gained in using the simulator to write sample ACS flight code, a methodology was developed to help write flight code and prepare the simulator once the ACS processor is ready. This methodology helps reduce the amount of “throw-away” work: work that cannot be used either as flight code or as part of simulator once the ACS processor is connected as hardware-in-the-loop. Though the methodology is focused on simulating the ACS processor, it can be applied to any microsatellite processor with peripheral systems, eg. Star Tracker processor connected to a CCD camera. A flowchart of the methodology can be found in Figure 4.1.

1. Using empty SuperBlocks, do a basic modeling of the ACS system (processor, sensors, actuators, and all the links between the systems) on the simulator.

2. Start creating software emulations of the peripherals, starting with those that can be done using only SystemBuild blocks. Continue with the models that require some C code to develop. Prioritize writing any code that will be used as flight code on the peripherals (eg. control code on a reaction wheel). Link these emulations to an environment model so that actuators will affect the attitude of the satellite and sensors will observe the environment.
3. Once the peripheral emulations are complete, start writing code for the ACS SuperBlock. The code should focus on functions that interact with the peripherals (sensors & actuators), which in the case of the ACS SuperBlock involves attitude control code and all the software-to-software interfaces to the peripherals. Test the code using the simulation. Debugging and testing will be an easier process because of the use of a block-model simulator system.

4. If any peripheral hardware becomes available before the ACS processor is completed, insert the hardware into the simulation and compare its behavior to its software emulation. Update the emulation if there are any significant differences. Remove the hardware from the simulation.

5. Repeat Step 3 if the ACS code has to be updated due to any changes to the peripheral emulations. Repeat Step 4 if any more peripheral hardware becomes available.

6. Once the actual ACS processor is ready, move all of the ACS SuperBlock code to the processor and connect it to the simulation system. The processor is now interacting with the software emulation of all its peripherals. Now that the connection between the ACS processor and the peripherals is a hardware-software connection, the interfaces to the peripheral emulations will have to be replaced.

7. This system is now the basis for a microsatellite command verification facility (CVF), a support tool for the operation of the microsatellite. It can be used to test changes to the flight code or new attitude control algorithms before uploading them to the microsatellite. As other processors become available (e.g. Housekeeping, Science), they can be connected to the ACS processor. The functionality of the entire microsatellite system can now be tested, with the simulation system taking the place of the environment, sensors, and actuators.

4.2. Work Efficiency Trade Study on Early Flight Code and Simulator Development

An analysis was made on the ratio of "throw-away" work to useful work for the various software emulations of the MOST ACS simulation when it was being prepared for interfacing
with the ACS processor hardware (see Section 6.1). All the work done on the simulator system can be divided into two types: flight code development and simulator specific development. All flight code is useful work while simulator specific work can either be useful or “throw-away” work depending if it can be used on the engineering model described in Step 7 of the methodology. Figure 4.2 explains this breakdown.

**Figure 4.1: Simulation Development Methodology Flowchart**

- **Create simulation model of ACS System.** Start with software models of peripherals (sensors/actuators), an environment model, and a space dynamics model.
- **Any sensor/actuator hardware available?**
  - **Yes**
    - Connect to simulator as hardware-in-the-loop
    - Update actuator/sensor software model(s) based on characterization simulations with real hardware
    - Remove hardware and insert updated software model(s)
  - **No**
    - Write flight code on the simulator that interfaces with the sensors/actuators models. The sensors and actuators interact with the environment and space dynamics models
    - Introduce actual ACS processor as hardware-in-the-loop when ready. Move flight code developed onto the processor
    - Integrate simulator with command verification facility. Facility used to support microsatellite when its in orbit.

**Figure 4.2: Simulator Work Breakdown**

- **Flight Code Work**
  - Can be used either on the microsatellite or the engineering model
- **Simulator Specific Work**
  - Throw Away Work
  - Effort that cannot be used when the actual ACS processor is ready
Applying these definitions to the work done on RT-Lab, a trade study on work efficiency was done. Table 4.1 details by system how much of the work for each emulation was useful, in terms of the number of SystemBuild blocks, lines of code, and work time. The work efficiency ratio was also given for each system emulation, where

\[
\text{Work Efficiency Ratio} = \frac{\text{Useful Work}}{\text{Total Work}}
\]  

A work efficiency ratio approaching 0.0 indicates that almost the entire emulation is “throw-away” work, while a ratio approaching 1.0 means that most of the emulation can be used as part of the command verification facility.

The work efficiency ratio for every system emulation was 0.5 or more, which was very good. By focusing the model development on developing only ACS code that interfaced with the sensors & actuators, the amount of “throw-away” work was limited to blocks and/or code that interfaced the software ACS SuperBlock to the peripheral emulations. These are the software-to-software connections described in Step 3 of the methodology. These interfaces will have to be changed if the ACS code on the actual processor is to be linked to the peripheral emulations. Figure 4.3 illustrates this concept. All blocks and code in the software emulations that linked them to the ACS Processor SuperBlock had to be removed for eventual replacement with blocks and code that would make them compatible to the ACS processor hardware.

Software driver blocks that will link the emulations to the hardware interfaces on the Target node of the simulator were included in these changes.

<table>
<thead>
<tr>
<th>System</th>
<th>Useful Blocks</th>
<th>Useful Code</th>
<th>Useful Work (days)</th>
<th>Work Efficiency Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnetometer Model</td>
<td>1</td>
<td>0</td>
<td>0.4</td>
<td>0.500</td>
</tr>
<tr>
<td>Sun Sensor Model</td>
<td>1</td>
<td>0</td>
<td>0.4</td>
<td>0.500</td>
</tr>
<tr>
<td>Rate Sensor Models (x3)</td>
<td>3</td>
<td>0</td>
<td>1.1</td>
<td>0.500</td>
</tr>
<tr>
<td>Magnetorquer Models (x3)</td>
<td>11</td>
<td>0</td>
<td>4.0</td>
<td>0.786</td>
</tr>
<tr>
<td>Reaction Wheel Models (x3)</td>
<td>6</td>
<td>115</td>
<td>8.8</td>
<td>0.793</td>
</tr>
<tr>
<td>Environment Model</td>
<td>7</td>
<td>360</td>
<td>23.3</td>
<td>0.985</td>
</tr>
<tr>
<td>ACS Model</td>
<td>0</td>
<td>690</td>
<td>39.7</td>
<td>0.853</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>29</strong></td>
<td><strong>1185</strong></td>
<td><strong>77.6</strong></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.3: Simulator Interface Diagram

Simulator Link Diagram:

Before ACS Processor is Available

Before ACS Processor is Available

After ACS Processor is Available

Figure 4.4 shows the work efficiency ratio as a function of the complexity of each system emulation, which is in order from the top of Table 4.1 to the bottom. As described in Section 3.3, complexity reflected how long it took to create the emulation and the difficulty in creating the emulation. The three sensor emulations, being essentially the same in complexity, were placed in the plot as one data point. A trend was developed from the data points.

Figure 4.4: Work Efficiency Plot Based on MOST Simulation

Work Efficiency Ratio vs. System Emulation Complexity

System Emulation Complexity (Low to High)
The goal of efficient flight code and simulation development is to focus on creating emulations whose work efficiency ratio is within the top end of this trend curve. Contrary to what might be expected, as the emulations became more complex, they proved to be more efficient in their use. Though they took more time to develop, these high complexity emulations had more components that could be used in the CVF. The less complex emulations, though good enough for use in the development and testing of ACS flight code, required much redesign in order to be incorporated into the CVF.

Another work analysis was done on the simulation. This one studied the work efficiency in developing the entire simulation, rather than focusing on the work efficiency in the development of each separate system emulation. A new ratio was calculated, called the cumulative work efficiency ratio, where

\[
\text{Cumulative Work Efficiency Ratio} = \frac{\text{Cumulative Useful Work}}{\text{Cumulative Total Work}} \tag{4.2}
\]

This ratio is measured each time a new emulation is completed. If the ratio continues to rise after each stage, then the overall development of the simulation is being done efficiently. When the ratio begins to drop, this indicates that the simulation development is no longer being done efficiently. When this occurs, it is a good indication that simulation development should be halted at, or soon after, the current stage.

Table 4.2 shows the cumulative work (both total and useful) and cumulative work efficiency ratio at each stage of the development of the simulation. Any inconsistencies between the sums in this table and the data in Table 3.2 and 4.1 are due to rounding.

<table>
<thead>
<tr>
<th>Emulation</th>
<th>Cumulative Work (days)</th>
<th>Cum. Useful Work (days)</th>
<th>Cum. Work Efficiency Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnetometer Model</td>
<td>0.7</td>
<td>0.4</td>
<td>0.500</td>
</tr>
<tr>
<td>Sun Sensor Model</td>
<td>1.5</td>
<td>0.7</td>
<td>0.500</td>
</tr>
<tr>
<td>Rate Sensor Models (x3)</td>
<td>3.7</td>
<td>1.8</td>
<td>0.500</td>
</tr>
<tr>
<td>Magnetorquer Models (x3)</td>
<td>8.8</td>
<td>5.9</td>
<td>0.667</td>
</tr>
<tr>
<td>Reaction Wheel Models (x3)</td>
<td>19.9</td>
<td>14.7</td>
<td>0.737</td>
</tr>
<tr>
<td>Environment Model</td>
<td>43.5</td>
<td>37.9</td>
<td>0.871</td>
</tr>
<tr>
<td>ACS Model</td>
<td>90.0</td>
<td>77.6</td>
<td>0.862</td>
</tr>
</tbody>
</table>
The cumulative work efficiency ratios were plotted against the cumulative total work and a trend was developed from the data points (see Figure 4.5). The trend showed that the cumulative work efficiency ratio kept on increasing as the development of the simulation continued, though the rate of increase approached zero by the time the ACS module was developed. Therefore, the overall simulation development, along with the development of each system emulation, was done in an efficient manner. Sample flight code was developed quickly and a majority of components in the simulation can be used in the CVF. However, there was a very slight drop of 0.09 (see Table 4.2) in overall work efficiency when the basic ACS flight code was finished. This is a warning that if more highly complex emulations and flight code are developed, then the overall simulation development might become inefficient.

*Figure 4.5: Cumulative Work Inefficiency Plot Based on MOST Simulation*

4.3. Flight Code Development Limitations

With these efficiency curves developed, the next step was to study the possibility of developing more flight code requiring even more complex software emulations of the ACS
Would the emulation efficiency curve in Figure 4.4 remain above 0.5, or would it drop above that level? Would the cumulative efficiency curve in Figure 4.5 remain high, or would it begin to drop? Two types of high complexity flight code development were studied: intra-processor code and inter-processor code.

**Intra-Processor Code:** This flight code includes reading telemetry sensors placed on the processor board (temperature, power, voltage), memory access and storage, and some low-level software driver development. The amount of flight code needed to perform these functions tends to be small, around 10 lines of code each, for a total of around 30. However, in order to do any useful development work, it will require a low-level emulation of the ACS processor and its linkages to other devices on the processor board, such as the memory devices and telemetry sensors. Such processor simulations, based on experience, tend to require at least 100 to 200 lines of “throw-away” C code and would also need a few SystemBuild blocks to emulate the sensors. This results in a work efficiency ratio of 0.3 at best, and 0.15 at worse. The cumulative work efficiency ratio also begins to drop. Given the low work efficiency ratio, this type of flight code should not be written until the processor hardware is available.

**Inter-Processor Code:** This flight code includes all of the serial software drivers needed to communicate between the model ACS processor and a model of the Housekeeping (HK) processor. It also includes the application program interfaces (APIs) needed to create and decode serial packets and the code that uses the APIs to send commands and receive telemetry over the serial bus. The most important aspect of inter-processor communications that can be checked using the simulator is the timing of packet transmissions: response acknowledgements to commands and the handling of commands that time-out. An attempt was made to write code for inter-processor communication using the simulator since the serial packet APIs had been previously been written by other members of the MOST team, but the attempt was eventually abandoned after about 100 lines of code were written. It was proving too difficult to simulate the serial communication timers that controlled packet flow for each embedded processor of MOST. Without an accurate simulation, any of the application code written using the APIs would be suspect when used on the actual processors. It could all end up being “throw-away” work, which
would give a very low work efficiency ratio, possibly a ratio of zero. This would also guarantee that the cumulative work efficiency ratio would start dropping.

As model complexity continues to increase, it was found by the above extrapolation that the work efficiency ratio also dropped to a very low value, as is shown in Figure 4.6. The cumulative work efficiency ratio also began to drop visibly, as is shown in Figure 4.7. It is important to keep all flight code and simulator development within the maximum of the curves in order to get work efficiently done early in the life of the microsatellite and to have a good simulator system ready when the hardware is ready so that an engineering model test system can be easily created.

**Figure 4.6: Work Efficiency Extrapolation**

![Work Efficiency Ratio vs. System Emulation Complexity](image)

**Figure 4.7: Cumulative Work Efficiency Extrapolation**

![Cumulative Work Efficiency Ratio vs. Total Cumulative Work](image)
5. ACS Flight Code Algorithms and Tests

5.1. Detumbling Algorithm Using Magnetorquer Actuation

The Detumble ACS mode relied only on the magnetorquers and magnetometers. This mode was used to reduce the tumbling rates of the microsatellite to 0.25 deg/s after it was placed into orbit by the launcher. Though algorithms do exist that provide some position control while detumbling, they were not used in this simulation.

A simple B-dot control law was used. The dipole moment commands to the magnetorquers (in the body frame) were set such that:

\[ \mathbf{m} = -k_b (\mathbf{b}_o - \mathbf{b}_m) \]  

(5.1)

where \( \mathbf{b}_o = [b_{o1}, b_{o2}, b_{o3}]^T \) was the time derivative of the magnetic field observed by the magnetometers, \( \mathbf{b}_m = [b_{m1}, b_{m2}, b_{m3}]^T \) was the time derivative of the on-board modeled magnetic field of the Earth (inertial frame), and \( k_b \) was a suitable scalar constant such that dipole moment produced would not exceed the maximum capability of the magnetorquers. In this case, \( k_b = 50000 \). The value of \( \mathbf{b}_m \) was calculated each time step of the simulation in the ACS flight code using an equation similar to Equation 3.24 (See Section 3.1).

The variable \( \dot{b}_m \) in the control law was introduced to reduce the effect of the change in the observed magnetic field due to the microsatellite orbiting the Earth on the desired magnetorquer control moments. When the satellite was detumbled, then \( \dot{b}_m = \dot{b}_o \). This made the detumbling algorithm more efficient. On the simulator, the on-board modeled magnetic field was the same as the environment model magnetic field with a 5% error introduced.

5.2. Coarse Pointing Algorithm Using Reaction Wheel Actuation

The coarse pointing mode used the reaction wheels to point and hold the microsatellite with respect to the solar pointing reference frame. Therefore, the aperture of the telescope on MOST would point in the anti-solar CVZ direction. A PD control law was used to control the reaction wheels, such that the commanded reaction wheel torques in the body frame were
\[
g_c = K_p \left( \dot{\theta} - \Delta \theta_{bt} \right) + K_d \left( \phi - \Delta \omega_{bt} \right) \\
= K(\dot{x} - \Delta x_{bt}).
\]

(5.2)

where \(K_p\) and \(K_d\) were positive-definite diagonal 3x3 matrices, \(K = [K_p, K_d]\), and \(\dot{x} = \left[ \dot{\theta}, \phi \right]^T\).

The variable \(\dot{\theta} = [\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3]^T\) was the estimated Euler angles of the microsatellite principle axis with respect to the inertial frame and \(\dot{\phi} = [\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3]^T\) was the estimated rotation rate state of the microsatellite with respect to the inertial frame. The variable \(\Delta x_{bt} = [\Delta \theta_{bt}, \Delta \omega_{bt}]^T\) was the difference in orientation between and rate between the telescope frame and the inertial frame. It was used to correct the desired control torques so that the microsatellite would be pointing towards the anti-solar CVZ. The deltas were simply defined as \(\Delta \theta_{bt} = [0, 0, -\omega_y, I]^T\) and \(\Delta \omega_{bt} = [0, 0, -\omega_y, I]^T\), where \(\omega_y\) was the orbital frequency of the Earth around the Sun.

The estimated states generated by the Estimator flight code were calculated using a non-linear model of the system implementing a Kalman Filter. All torques except for those produced by the reaction wheels and the magnetorquers were unmodeled. The state equation of the estimator was

\[
\dot{x} = Ax + Bu + L(C(\dot{x}) - y) \\
u = -K(\dot{x} - \Delta x_{bt}) - \dot{\omega} \cdot h_w(t)
\]

(5.3)

where \(B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\), \(A = \begin{bmatrix} 0 & S^{-1}(\dot{\theta}) \\ 0 & -I^{-1}(\dot{\omega}^*I) \end{bmatrix}\), \(C(\tilde{x}) = \begin{bmatrix} C_{bt}(\tilde{\theta})b_{lm} \\ \dot{\phi} \\ C_{bt}(\tilde{\theta})C_{st}s_t \end{bmatrix}\). \(C_{bt}\) was the rotation matrix between the body axis frame and the inertial frame, \(b_{lm}\) was the on-board modeled magnetic field in the inertial frame normalized to a magnitude of 1.0, and \(s_t = [1 0 0]\) was the direction vector pointing towards the Sun in the solar pointing frame. The variable \(y\) was a 9x1 matrix containing the sensor outputs from the magnetometer (normalized to a magnitude of 1.0), rate sensors, and sun sensor respectively. In the case of the magnetometers, the outputs were normalized to 1.0.
\[ J = \frac{1}{2} \int_0^\infty (x^T Q x + u^T R u) \, dt \]  

(5.4)

This can be solved by determining a positive-definite solution of \( X \) in the Riccati equation

\[ X A_{\text{lin}} + A_{\text{lin}}^T X - X (B R^{-1} B^T) X + Q = 0 \]  

(5.5)

where \( K = R^{-1} B^T X \), \( A_{\text{lin}} = A_{\xi=0} \)

The variables \( Q \) (6x6 positive semi-definite matrix) and \( R \) (3x3 positive definite matrix) were weighting matrices, selected so that the maximum applied torques did not exceed 0.003 Nm, the maximum allowable on the Dynacon reaction wheel. Solving for \( K \) led to the following value for use in the PD controller of the ACS flight code

\[
K = \begin{bmatrix}
0.0032 & 0 & 0 & 0.0796 & 0 & 0 \\
0 & 0.0032 & 0 & 0 & 0.0796 & 0 \\
0 & 0 & 0.0032 & 0 & 0 & 0.0796
\end{bmatrix}  
\]  

(5.6)

A modified version of the LQR method was used to determine \( L \). The performance function used here was the same as before, except \( R \) was now a 9x9 matrix. The Riccati equation being solved now had the form

\[ Y A_{\text{lin}}^T + A_{\text{lin}} Y - Y (C_{\text{lin}}^T R^{-1} C_{\text{lin}}) Y + Q = 0 \]  

(5.7)

where \( L = -R^{-1} C_{\text{lin}}^T Y \) and

\[
C_{\text{lin}} = \frac{\partial C}{\partial x} \bigg|_{\xi=0} = \begin{bmatrix}
\frac{\partial C_{\text{bl}}}{\partial \theta_1} b_{\text{ln}} & \frac{\partial C_{\text{bl}}}{\partial \theta_2} b_{\text{ln}} & \frac{\partial C_{\text{bl}}}{\partial \theta_3} b_{\text{ln}} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & -b_{\text{lm}3} & b_{\text{lm}2} & 0 & 0 & 0 \\
b_{\text{lm}3} & 0 & -b_{\text{lm}1} & 0 & 0 & 0 \\
-b_{\text{lm}2} & b_{\text{lm}1} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
In Equation 5.3, L multiplied the difference (or error) between \( C(\hat{x}) \) and y. Therefore, the differential matrix \( \frac{\partial C}{\partial x} \big|_{x=0} \) was used to optimize L.

However, \( b_{li} \), and hence \( C_{li} \), was dependent on the position of the microsatellite in its orbit as well as the rotation of the Earth on its own axis. One method for accommodating this would be to recalculate the \( C_{li} \) matrix and solve Equation 5.8 every simulation time step to determine the optimized value for L. However, this would be computationally intensive and would lead to much complication. Another method would be to switch everything into a discrete time format to solve for L. Such a format would lead to equations that are less complicated to solve. In the end, another solution was discovered.

The matrix L was calculated for the position of the microsatellite at the simulation start time (Vernal Equinox, true anomaly =0.0). A simulation was then executed with the estimator only using this L matrix. As shown in Figure 5.1, the microsatellite managed to make it through about one-sixth of its orbit (1000 s) before its position control algorithm no longer worked because the state estimator became unstable. This L matrix was no longer ideal for the current position of the microsatellite. The sidereal rotation of the Earth had no appreciable effect on how far the microsatellite could travel in its orbit; the near-symmetry in the dipole magnetic field model would account for such a small influence. Therefore, if the estimator flight code had another L matrix that was calculated for an orbital position with a true anomaly of 45° and a simulation time of 0.0, it could switch to using that value of L once the on-board orbit propagator approached the halfway point between the two positions (22.5°).

Assuming the on-board orbit propagator never diverged greatly from the actual orbit of the microsatellite (<10% error), a series of L matrices could be calculated to cover the entire orbit, one for every eighth of the orbit. The actual (non-normalized) \( b_{li} \) vectors for all eight positions are listed in Table 5.1. Due to symmetry, there were only four different \( b_{li} \) vectors. Hence, there were only four difference \( C_{li} \) matrices and only four L matrices had to be calculated and stored in memory with the estimator flight code. The values of these L matrices can be found in the flight code file “usrestimator.c” in Appendix D. When running, the estimator switched from
one L matrix to another when the on-board orbit propagator reached the halfway point between two of the true anomaly values tabulated in Table 5.1.

![Figure 5.1: Destabilization of Coarse Pointing Control Due to Changing Magnetic Field](image)

<table>
<thead>
<tr>
<th>True Anomaly</th>
<th>$b_{lm1}$</th>
<th>$b_{lm2}$</th>
<th>$b_{lm2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0^\circ, 45^\circ$</td>
<td>3.906144e-6</td>
<td>-7.787243e-6</td>
<td>2.1099040e-5</td>
</tr>
<tr>
<td>$45^\circ, 225^\circ$</td>
<td>-1.711708e-6</td>
<td>-3.4113787e-5</td>
<td>-1.6490888e-5</td>
</tr>
<tr>
<td>$90^\circ, 270^\circ$</td>
<td>-5.603018e-6</td>
<td>3.893620e-6</td>
<td>-4.2528259e-5</td>
</tr>
<tr>
<td>$135^\circ, 315^\circ$</td>
<td>1.4834e-8</td>
<td>3.0220166e-5</td>
<td>-4.938332e-6</td>
</tr>
</tbody>
</table>

5.3. Reaction Wheel Desaturation Algorithm Using Magnetorquer Actuation

The magnetorquers were also used to reduce the angular velocities of the reaction wheels to prevent them from becoming saturated. Many schemes have been developed to perform reaction wheel momentum management (see [7]); the one used in the simulation was the conventional cross-product law (CCPL). The dipole moment commands to the magnetorquers are set such that

$$ m = \frac{k_m (\mathbf{h}_{wd}(t)b_o)}{\| b_o \|} $$

$$ h_{wd}(t) = I_w (\omega_w(t) - \omega_d) $$

(5.8)
where \( k_m \) was a positive scalar constant and \( \omega_d = [\omega_{d1} \omega_{d2} \omega_{d3}]^T \) were the desired wheel speeds in rad/sec. The variable \( k_m \) had a value of 5.0 A-m/N in the simulation. If the value was much larger, the microsatellite would become unstable as the reaction wheels desaturated because the torques produced by the magnetorquer would be too large for the state estimator to handle effectively.

5.4. ACS Simulation Test Results

Figure 5.2 shows the results of one simulation using the detumble control algorithm, with initial rotation rates of \( 10^\circ/s \) with respect to the inertial reference frame. With an orbital period of 6033 s, the rotational rates were damped to reasonable levels after one third of an orbit.

![Figure 5.2: Detumble Experiment Results](image)

ACS B-Dot Detumble Results (Principal Axis Frame wrt Inertial Frame)

The results of a successful simulation run using the coarse pointing and desaturation control algorithms can be seen in Figures 5.3 and 5.4. The initial conditions of the microsatellite are \( 10^\circ \) for each position angle and \( 0.25^\circ/s \) for each rotation rate. The coarse pointing algorithm is started at 0 s and is followed by a response from a user-created disturbance response at 500 s. The spikes in Figure 5.4 at 500 s are the reaction wheels responding to the disturbance and trying to keep the spacecraft pointing in the desired direction. This causes a momentary spike in the
angular position of the spacecraft at the same time (see Figure 5.3). These spikes take about 150 sec. to settle down. The reaction wheels are then desaturated starting at 1200 s. The desaturation algorithm stays active until the end of the simulation. When started, the desaturation algorithm causes a spike in the angular position of the spacecraft at 1200 s (see Figure 5.3). Body Axis 1 and Axis 2 quickly settle down into their proper pointing positions in about 500 s, but it takes 1200 s for Body Axis 3 to settle down. The hardware reaction wheel is desaturated to 10 rad/s rather than 0 rad/s because it has problems controlling its speed when it is below 5 rad/s (see Figure 3.8 in Section 3.1).

Figure 5.3: Coarse Pointing/Desaturation Experiment Results (Angular Position)

![Graph showing Angular Position over time](image)

Figure 5.4: Coarse Pointing/Desaturation Experiment Results (Reaction Wheel Rates)

![Graph showing Reaction Wheel Rates over time](image)
6. Future Expansion of MOST Simulator

6.1. MOST Command Verification Facility (ACS Processor as Hardware-in-the-Loop)

With the MOST ACS Flight Code simulation completed and most of the sample ACS flight code written, the next two steps as outlined in the methodology prepared the simulator to become the Command Verification Facility (CVF) for the MOST microsatellite. This facility will initially be used to test the flight code developed on the simulation using the ACS processor hardware. The code will require some changes to accommodate the hardware architecture of the new processor, however these changes will be at a minimum. The simulator would not allow the use of any unusually complex C commands because it only used the old WATCOM C v5.1 compiler. Therefore, the C functions used in the flight code should work on the processor and require no changes.

Figure 6.1 shows the configuration of the CVF. The ACS processor hardware running the flight code, including the on-board orbit propagator copied from the environment emulation code, is connected as hardware-in-the-loop to the simulator. In essence, the ACS processor hardware takes the place of the ACS Processor SuperBlock and the Torque blocks in the Actuator SuperBlock. When the facility is running, the ACS processor gives actuator commands to the simulator via the hardware interfaces on the Target node. The simulator runs the actuator, sensor, and environment software emulations as before, and all sensor observations, including the serial packet responses from the reaction wheels, are sent back to the ACS processor for state estimation and telemetry recording. The ACS processor can also be connected to other available microsatellite hardware, such as the House Keeping processor, telecommunication & telemetry control (TT&C) nodes, and the Science CCD processor. See Figure 3.1 for other examples of hardware that can be included. Unlike the sensors and actuators, such hardware does not require any specialized ground support equipment and thus can be connected to the CVF via the ACS processor. Including such extra hardware allows the user to test more of the functionality of the ACS processor while it interacts with its simulated sensors, actuators, and environment.
As described in Section 4.1 and shown in Figure 4.3, the simulator must be modified to remove the "throw-away" work. This work included any blocks or code that dealt with the software-to-software links between the ACS Processor SuperBlock and the sensor & actuator emulations. They had to be replaced with code and blocks that run the software drivers linking the emulations to the hardware interfaces on the Target node, creating the hardware-to-software link between the ACS processor hardware and the sensor & actuator emulations. Along with these changes, some others had to be made. In the MOST Sample ACS Flight Code Development simulation, the sensors provided their observations to the ACS Processor SuperBlock in their actual units (e.g. the magnetometer outputted its results in H), rather than in volts or current as in reality. The magnetorquers also received their commands in desired magnetic moment rather than current. This was done because when the original simulation was developed, the exact hardware that was going to be used was not known. Therefore, it was impossible to model the voltages and currents that this hardware would require. Since the rate sensors on MOST are part of the Dynacon reaction wheel package, it was also decided to move their emulations over to the reaction wheel emulations at this stage.
All the modifications required to prepare the simulation for the CVF were made. As of this time, the ACS processor has not yet been available for connection to the simulator as hardware-in-the-loop. As will be shown, some calibrations will still have to be made to the sun sensor and magnetorquer emulations to make sure they interact properly with the ACS processor.

6.1.1. Changes Made To Create MOST CVF Simulator

Though all of the SuperBlocks in the simulation were modified, many of the modifications only involved the removal of input/output connections between blocks, due to the removal of the ACS Processor SuperBlock. The changes that are highlighted in this section are those that involved the addition or removal of blocks to a specific SuperBlock.

One general change made to the simulator was the change of the inertia matrix so that it was a closer match to the current value for MOST. When the simulation was originally made, this inertia matrix had yet to be calculated accurately for MOST, thus values of the same order of magnitude were used instead. Table 6.1 shows the principal moments of inertia around the centroid of MOST, along with the directions of the principal moments. Using these eigenvectors, the direction cosine between a vector rotated to this frame and the original vector in the body frame was calculated. The angle between these vectors was 5.7°. This angle was very small and justified the original assumption made in Section 3.1 that the body frame was the same as the principal axis frame. If so desired, a rotation matrix can be introduced into the Attitude SuperBlock to compensate for this slight offset.

<table>
<thead>
<tr>
<th>Principal Moments of Inertia (kg-m^2)</th>
<th>Direction of Principal Axes ([x y z] Body Frame)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8</td>
<td>[0.9995, -0.0195, 0.0231]</td>
</tr>
<tr>
<td>1.8</td>
<td>[0.0210, 0.9975, -0.0675]</td>
</tr>
<tr>
<td>1.5</td>
<td>[-0.0217, 0.0680, 0.9975]</td>
</tr>
</tbody>
</table>
Table 6.2 lists the system emulations that were modified and how many new blocks and lines of code were required to make those modifications. Using the same time estimates applied in Section 3.3, an estimate on the total work time required to make these changes was made. It took less than a week and a half to make all the necessary modifications, which helped to verify the time estimate assumptions made in Section 3.3.

Diagrams of each modified SuperBlock can be found in Section II of Appendix C.

<table>
<thead>
<tr>
<th>System Emulation</th>
<th>New Blocks Required</th>
<th>New Lines of Code Required</th>
<th>Approx. Work (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnetometer Model</td>
<td>5</td>
<td>0</td>
<td>1.8</td>
</tr>
<tr>
<td>Sun Sensor Model</td>
<td>11</td>
<td>0</td>
<td>4.0</td>
</tr>
<tr>
<td>Magnetorquer Models (x3)</td>
<td>1</td>
<td>0</td>
<td>0.4</td>
</tr>
<tr>
<td>Reaction Wheel Models (x3)</td>
<td>8</td>
<td>0</td>
<td>2.9</td>
</tr>
<tr>
<td>Environment Model</td>
<td>0</td>
<td>16</td>
<td>0.9</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>10.1</td>
</tr>
</tbody>
</table>

**Orbit SuperBlock**

*Master → Environment → Orbit*

The changes made here dealt with the sun position model. The field of view of the sun sensor was now known to be ±67°, thus the sun would be not be visible to the sun sensor if the aperture of the sun sensor was not within these limits due to the angular position of the microsatellite. The sun model was modified to take this into account. The rotation blocks for the sun direction vector in the body frame were also removed due to a change in the data being sent to the sun sensor emulation. Rather than sending the sun vector components to the sensor emulation, it must now send the two angles by which the face of the sun sensor (on the negative x-axis face) was offset with respect to the direction of the sun. In the terminology used by the sun sensor, these two angles, θ_{b2} and θ_{b3}, were the elevation angle and the azimuth. Using C code rather than blocks, the rotation matrices \( C_{rl} \) and \( \tilde{C}_{bl} = C_2(\theta_2)C_3(\theta_3) \) were used to rotate \( \mathbf{s}_r \) (sun vector in telescope frame) into the body frame \( \mathbf{s}_b = \begin{bmatrix} s_{b1} & s_{b2} & s_{b3} \end{bmatrix} \). The matrix \( \tilde{C}_{bl} \) was
used instead of $C_{bl}$ because the sun sensor has no way of determining the value of $\theta_1$ and thus it must be removed from all calculations. The angle $\theta_{b2}$ was then determined using the following

$$\theta_{b2} = (-\text{sign}(s_{b3})) \cdot \cos \left( \frac{s_{b1}}{\sqrt{s_{b1}^2 + s_{b2}^2}} \right)$$

$$= (-\text{sign}(s_{b3})) \cdot \cos \left( \frac{s_{b1}}{\sqrt{s_{b1}^2 + s_{b2}^2}} \right)$$

(6.1)

The angle was either positive or negative depending on the sign of $s_{b3}$. The vector $s_b$ was then rotated by the matrix $C(\theta_{b2})$ and took on the value $\tilde{s}_b = [\tilde{s}_{b1} \tilde{s}_{b2} \tilde{s}_{b3}]$. The angle $\theta_{b3}$ was then calculated

$$\theta_{b3} = (-\text{sign}(\tilde{s}_{b3})) \cdot \cos \left( \frac{\tilde{s}_{b1}}{\sqrt{\tilde{s}_{b1}^2 + \tilde{s}_{b2}^2}} \right)$$

$$= (-\text{sign}(s_{b3})) \cdot \cos \left( \frac{\tilde{s}_{b1}}{\sqrt{\tilde{s}_{b1}^2 + \tilde{s}_{b2}^2}} \right)$$

(6.2)

These equations were derived using the inverse of the cosine angle law

$$\cos (\text{angle between } a \text{ and } b) = \frac{a^T b}{\|a\| \|b\|}$$

In the case of both equations, the matrix $[1 0 0]^T$ was $s_t$. In Equation 6.1, the angle between $s_b$ and $s_t$ around Axis 2 of the solar pointing frame (sun sensor elevation) was desired. After rotating $s_b$ around this Axis 2 by $\theta_{b2}$, the angle between $s_b$ and $s_t$ around Axis 3 of the solar pointing frame (sun sensor azimuth) was determined in Equation 6.2. The functionality of these equations was verified using Matlab.

These two angles, along with the visibility status of the Sun, were sent to the sun sensor emulation. The matrix $\hat{C}_{bl}$ can also be used in the state estimator flight code to determine the direction of the Sun using the estimated states.
Satellite SuperBlock

*Master → Satellite*

The biggest change to this SuperBlock was the removal of the ACS Processor SuperBlock; all of the ACS functionality was handled by the processor hardware. The outputs from the Sensor SuperBlock were the magnetometer and sun sensor telemetry being sent to the Console SuperBlock for display. The blocks handling the sensor outputs to the processor hardware interfaced to the simulation were placed in the Sensor SuperBlock. The inputs into the Actuator SuperBlock were the body frame magnetic field values and the rates of spin of the microsatellite, which came from the Environment SuperBlock. These values are required for the proper operation of the magnetorquer and rate sensor emulations.

Sensor SuperBlock

*Master → Satellite → Sensor*

The changes made to this SuperBlock were the removal of the Rate Sensor SuperBlock, which was moved to the reaction wheel emulation, and the addition of the Sensoray 626 analog output driver block. The RT-Lab simulator had two Sensoray 626 boards, each with four analog output pins. The outputs from the Sun Sensor emulation and magnetometer emulation each went to a different board.

In the Properties window of the Sensoray analog output blocks, the user must define in the Integer Parameters section which board is being used. This parameter ranges from 0 to $n$ where $n+1$ is the number of Sensoray board available. The magnetometer outputs used Board 0 and the sun sensor outputs used Board 1. The Sensoray analog output box also allowed the user to send different outputs when the simulator was in one of three execution modes: Reset, Pause, or Run mode. For this case, the same set of outputs was used for all three modes.

It was discovered that there were discrepancies between the desired output values and the actual values outputted from the Sensoray 626. The reason for these discrepancies was never discovered. However, through experimentation, the value of the discrepancy for each output pin
was found to stay constant down to the cV and thus it could be compensated though the addition of two extra blocks.

**Magnetometer SuperBlock**

*Master → Satellite → Sensor → Magnetometer*

Information about the magnetometer used by the MOST microsatellite can be found in Appendix B. The magnetometer used on MOST sends it observations to the ACS processor in the form of a voltage, which is then converted by the flight code to a magnetic field reading. The conversion equation is

$$b_m = (v_m - z_m) \left( \frac{10^{-6}}{25 \times 10^{-6}} \right) \text{Tesla}$$

(6.3)

where $v_m$ contained the three voltage outputs of the magnetometer and $z_m$=[2.5V 2.5V 2.5V] was the zero-point of the magnetometer. Equation 6.3 was placed into the sample ACS flight code for use by the estimator and the detumbling algorithm. The modified magnetometer emulation, of course, did the reverse of Equation 6.3 so that the proper voltage could be sent to the ACS processor.

The maximum magnetic reading the sensor can detect is higher than the maximum value of the Earth's magnetic field at 785 km. Since the maximum voltage the magnetometer can output was less than 10 V, the maximum analog output value of the Sensoray 626, there were no difficulties making this emulation work with the ACS processor. As will be shown, this was not always the case for other sensor and actuator emulations.

**Sun Sensor SuperBlock**

*Master → Satellite → Sensor → Sun Sensor*

Information about the sun sensor used by the MOST microsatellite can be found in Appendix B. Essentially, the sun sensor consists of four photodiodes each placed in one quadrant of the sensor. Sunlight enters the aperture of the sun sensor and the direction of the sun
is determined based on the four currents running through the diodes. The Cartesian coordinates of the sunlight on the sensor face was determined by

\[
X = \left( \frac{(Q2 + Q3) - (Q1 + Q4)}{Q1 + Q2 + Q3 + Q4} \right) \\
Y = \left( \frac{(Q1 + Q2) - (Q3 + Q4)}{Q1 + Q2 + Q3 + Q4} \right)
\] (6.4)

where Q1, Q2, Q3, and Q4 are the 4 photodiode currents, and X and Y are always between ±1.0.

From this, two of the angles of rotation of the sun sensor, and hence the microsatellite, were determined using the following equations

\[
\theta_{b2} = A_o + A_x \cdot X + A_y \cdot Y \\
\theta_{b3} = B_o + B_x \cdot X + B_y \cdot Y
\] (6.5)

The direction of the sun in the frame of the microsatellite was then determined using the matrix

\[ C_{bt} = C_2(-\theta_{b2})C_3(-\theta_{b3}). \]

Equations 6.4 and 6.5 are standard for the AeroAstro sun sensor. The constants in Equation 6.5 are determined by AeroAstro through experimentation for each sun sensor. The constants A_o, B_o, A_y, and B_x tend to be very small and are equal to 0.025, 0.039, 0.001, and 0.005 respectively for the sun sensor on MOST. They can be changed in the future if any new calibration tests show that the values have changed. The constants A_x and B_y were equal to 0.565 and -0.564 respectively. This means that the maximum angles from which the sun sensor could detect the sun accurately (within 1°) were ±31° for both \( \theta_{b2} \) and \( \theta_{b3} \). Though the field of view of the sun sensor was ±67°, the accuracy of the sun sensor degraded as the angles exceeded the ideal position. Equations 6.4 and 6.5 were placed into the sample ACS flight code for use by the estimator.

The modified sun sensor emulation had to do the reverse of these equations so that the proper photodiode currents were sent to the ACS processor. Values for \( \theta_{b2} \) and \( \theta_{b3} \) were taken from the Orbit SuperBlock and saturated at a value of ±31° to simulate the degrading accuracy of the sun sensor when the angles exceeded this range. In addition, a line of code was added to the sun position emulation in the Orbit SuperBlock so that the sun sensor was shut off (similar to
when the microsatellite was in eclipse) when either $\theta_{b2}$ or $\theta_{b3}$ exceeded $\pm 67^\circ$. When the sun sensor was considered shut off (an input of 0 into the switch block), the direction of the sun defaulted to $[1 \ 0 \ 0]^T$ like before. This will have to be changed once it is known how Dynacon deals with the sun sensor outputs when the Sun is not visible. If the sun sensor was on, X and Y were then calculated using $\theta_{b2}, \theta_{b3}$, and the following equations

$$
X = \frac{(\theta_{b2} - A_0) \left( \frac{By}{Ay} \right) - (\theta_{b3} - B_0)}{(By \cdot Ax - Ay \cdot Bx)} Ay \\
Y = \frac{(\theta_{b3} - B_0) \left( \frac{Ax}{Bx} \right) - (\theta_{b2} - A_0)}{(By \cdot Ax - Ay \cdot Bx)} Bx
$$

(6.6)

A problem in creating the emulation then occurred. In order to determine the four current values for a specific X and Y, Equation 6.4 had to be used. However, there were only two equations with four unknowns to be solved, thus there was no independent solution. A least-squares algorithm that guarantees non-negative solutions for all for currents could have been used to determine a solution. However, the equations are non-linear and such an approach would have been computationally intensive. A better approach was to define a third equation specifying the total of all four currents

$$Q1 + Q2 + Q3 + Q4 = T$$

(6.7)

The value of T chosen had no effect on the sun sensor equations, since it was the relative values of the currents that were necessary in deriving the orientation of sun sensor. At this time, AeroAstro had yet to specify the magnitude of the typical currents the sun sensor would provide, thus it was decided to set $T = 0.01$ A temporarily. The value of the total current can be changed in the future when more is known about the sun sensor. However, in the end, it only matters that the value of T chosen is greater than 0 and less than the input tolerances of the ACS processor board.
With the addition of $T$, the following equations were derived

$$Q_1 = \frac{(Y + 1)T}{2} - Q_2$$
$$Q_2 = \frac{(X + Y)T}{2} + Q_4$$
$$Q_3 = \frac{(X + 1)T}{2} - Q_2$$ \hspace{1cm} (6.8)

With the constraints that $T > 0$ and $-1 > X, Y > 1$, then the following steps can be used to guarantee that none of the currents become negative:

1) If $(X + Y > 0)$ then $Q_4 = 0$ and solve Eq. 6.8
2) If $(X + Y < 0)$ then $Q_2 = 0$ and solve Eq. 6.8
3) If $(X + Y = 0)$ then $Q_2 = Q_4 = 0$ and solve Eq. 6.8

Following these steps gave the same results as doing a least-squares analysis with non-negative solutions on Equations 6.4 and 6.7.

One problem with connecting this emulation to the ACS processor is that while the outputs from the Sensoray 626 analog ports are in volts, the ACS processor needs the sun sensor telemetry to be in amps. A voltage controlled current source will have to be inserted between the simulator and the ACS processor to guarantee that a unique voltage command from the analog port will give a unique current that the ACS processor can detect and process correctly. Again, it is not necessary that the four currents provided by the emulation exactly match those that would be given by the actual sun sensor for a given orientation. It is only required that the magnitude of the currents provided by the simulator do not exceed the tolerances of the ACS processor and that they do not become negative.
Magnetorquer SuperBlock

Master → Satellite → Actuator → Magnetorquer

Information about the magnetorquers used by the MOST microsatellite can be found in Appendix B. Each magnetorquer on MOST can create a maximum magnetic moment of about 5 A-m², and the voltage required to produce that moment is 5 volts. A voltage of -5 volts will produce a 5 A-m² moment in the opposite direction.

The only change required for the magnetorquer emulation was the insertion of a Sensoray 626 analog input driver block. However, there are going to be some issues when interfacing this emulation to the ACS processor. The processor gives its magnetic moment commands in the form of a current rather than a voltage. Therefore, a current controlled voltage source will have to be connected between the simulator and the ACS processor to convert the command current into a voltage.

Reaction Wheel SuperBlock

Master → Satellite → Actuator → Reaction Wheel

The serial driver blocks used were similar to those used in the Hardware Reaction Wheel SuperBlock (see Section 3.1.1), though all 27 packet bytes (9 for each reaction wheel) are sent in one transmission. The one difference was which serial ports on the simulator were used. At the time these changes were made, the interrupt capabilities on the IP501 board were no longer functioning properly due to some problems with the hardware. Therefore, the built-in serial ports of the Target computer, connected to an RS-232-to-RS-422 converter, were used instead.

While the MOST project proceeds, it is most likely that the serial packet format used to communicate with the reaction wheels will change to accommodate the Simple Serial Packet (SSP) protocol (see Appendix B) used by the on-board computers (HK, ACS, Science, Star Tracker) to communicate with each other. The only changes to the simulation that will have to be made are to the IP501 driver blocks, which must now handle a larger serial packet, and to the C code in the reaction wheel emulation, so that it can properly extract the reaction wheel.
command from the SSP packet. SSP will make the extraction of telemetry from the reaction wheel more flexible and thus will remove the need to have the C code that estimated the torque of the wheels.

**Software Reaction Wheel SuperBlock**

*Master → Satellite → Actuator → Software Reaction Wheel*

The only change to this SuperBlock was the inclusion of the Rate Sensor SuperBlock, moved here from the Sensor SuperBlock. Since the rate sensor is part of the Dynacon reaction wheel package, this emulation was a better match to the actual flight hardware. The ACS processor received telemetry from the rate sensor via the 9-byte response packet of the reaction wheel. The ACS processor only had to request the rate sensor telemetry using the 9-byte command packet.

**Console SuperBlock**

The ACS processor, via user command through the HK computer, now controlled the ACS mode of the microsatellite in the simulation. Therefore, the multiplexer box, which allowed the user to control the ACS mode via the simulation Console, was removed. The user could no longer control the reaction wheels either. Therefore, three separate Primary Axis SuperBlocks were no longer needed and all of the microsatellite telemetry was displayed in one SuperBlock. The only inputs the user could give to the simulation via the Console display were disturbance torques. The associated SuperBlock remained the same.

**Telemetry SuperBlock**

*Console → Telemetry*

The new Telemetry SuperBlock displayed all of the raw sensor data from the magnetometer (in volts) and the sun sensor (in amps). Other sensor data included the speeds of the reaction wheels. The status of the orbital environment (e.g. position) along with telemetry
about the microsatellite (e.g. position angles, rates, magnetic torques, reaction wheel torques) was also reported.

6.1.2. Note on CVF Development

In was necessary to test the simulator while it was being changed to make sure the sensors and actuators still worked properly. As well, the ACS processor was unavailable at the time the changes were made, so a scheme had to be developed to test the hardware interfaces to make sure they were working properly. To this end, the sensor emulation analog outputs sent to the Target node hardware interfaces (where the ACS processor hardware will eventually be connected) were directly connected to the analog inputs of the Target node. These inputs were then sent to the ACS Processor SuperBlock, which was temporarily retained. In essence, a software-hardware-hardware-software connection was made. See Figure 6.2 for a diagram of this setup in the Sensor SuperBlock. The serial input and output lines for the reaction wheels were also directly connected to each other in a similar manner. There were not enough D/A ports available to connect the magnetorquers in this manner. Once the ACS processor is available, the ACS SuperBlock is removed and the simulation along with all the hardware connections feeding back into the simulator, as described in Section 6.1.1. These connections are then interfaced with the ACS processor as designed.

Figure 6.2: Software-Hardware-Hardware-Software Connection SuperBlock
In hindsight, a simulation could incorporate this design technique from the start. If the exact nature of the hardware connections and command formats between the ACS processor and the sensors & actuators are known before design of the simulation is started, then this design technique could be used. In the case of MOST, most of this information was not known when simulation design began. Therefore, any time spent trying to create these software-hardware-hardware-software connections would be wasted as "throw-away" work, work that could have instead been focused on creating flight code. Since this lack of information would be standard for most startup microsatellite projects, it was decided not to include this design technique in the simulation design methodology described in Section 4.1. It is mentioned here only as a note of interest.

6.2. Complete Microsatellite Simulator

The simulation designed here focused on the ACS subsystem of the microsatellite. That was because the whole point of designing the simulation was to potentially help develop ACS flight code and a command verification facility to support the operation of the microsatellite. However, in the future, it might be useful to have a microsatellite simulation that emulates more aspects of the spacecraft. Such as system could be used for future academic research as well as a tool for the project development of new microsatellite projects. Such a simulator would be similar to what was done at the Harbin Institute of Technology (see Section 1.2.3). As shown in Figure 1.5, a complete microsatellite simulation would include such systems as thermal and power inclusive with an attitude control system. However, trying to develop such a complete simulation on the one-node RT-La used here for MOST might not be possible due to the difficulty if getting such a complex real-time, hardware-in-the-loop simulation working efficiently on one target node.

As mentioned in Section 2.3, the Space Dynamics Lab at UTIAS purchased a multiple Target node RT-La simulation system. See Figure 2.4 for a diagram of the system. The ACS simulation developed for MOST could be copied over to this system and form the core of a complete microsatellite simulation. This simulator can be used not only to test out new
microsatellite designs and develop flight code, but can be used to help develop new microsatellite hardware.

Along with the RT-Lab multi-node simulator, the Space Dynamics Group will have access to a three-axis air-bearing table courtesy of SFL and a Helmholtz magnetic chamber. The magnetic chamber can be connected to the simulator as hardware-in-the-loop to provide magnetic fields matching those of an orbital environment model running in a simulation. This system can be used to develop newer and better magnetorquer control actuators. Newly designed reaction wheels, control moment gyros (CMGs), and rate sensors can be interfaced to the simulator and placed on the air-bearing table. These actuators and sensors can then be tested so that their functionality can be determined and their design improved.

All of this is pure speculation. Unlike the CVF, no work has yet been done in implementing this simulation design. However, all of the simulator equipment is available, so it is just a matter of time and work to implement these ideas.
7. Summary & Conclusion

In Section 1.4, a list of objectives was presented to determine and develop strategies that could be used to perform efficient and concurrent microsatellite software and hardware development through the use of a real-time simulator. The goal was to determine if such simulation systems could prove to be beneficial if used in such a manner. The following is a summary of how those objectives were met and what final conclusions were drawn from the research work performed.

Using the Opal-RT RT-Lab real-time, hardware-in-the-loop simulator, a simulation of the ACS system of MOST was created. At this time, the ACS processor hardware for MOST was yet to be completed. Emulations of all the ACS sub-systems were made and the simulation also included an orbital environment model. The reaction wheel emulation was refined using the hardware reaction wheel. It was temporarily interfaced to the simulator as hardware-in-the-loop and its performance characteristics were recorded and modeled in the emulation. Using the simulation, sample ACS flight code was written that could be used on MOST. This flight code was created concurrent to the development of the ACS hardware, thanks to the ability of the simulator to emulate the hardware. The flight code can potentially be transferred to the ACS processor with little modification. The simulation was then modified so that the ACS processor could be interfaced as hardware-in-the-loop. Once MOST is launched, this new configuration of the simulator can be used as a command verification facility (CVF) to test new or modified flight code before it is uploaded and executed on MOST.

This simulator development work, including writing the sample flight code and the modifications to create the CVF, took four months to complete. This quick development time was due, in part, to the use of SystemBuild, which made it possible to quickly create emulations with out the need to write any C code. Using the experience gained from doing this work, a simulation design methodology was developed (Section 4.1) to help minimize “throw-away” work, to maximize the amount of flight code that can be developed early, and maximize simulation work that could be used as part of the CVF. By using a work efficiency trade study of the simulation based on the methodology developed, it was determined what flight code can be developed early, and what flight code should be delayed until the processor hardware is ready.
(Table 7.1). The same trade study was also used to determine what hardware could be added to the microsatellite simulator once the ACS processor is available, and what hardware should be emulated (Table 7.2).

<table>
<thead>
<tr>
<th>Table 7.1: Flight Code Development Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early</td>
</tr>
<tr>
<td>ACS Comm. With Reaction</td>
</tr>
<tr>
<td>Actuator Torque Estimation</td>
</tr>
<tr>
<td>State Estimator/Kalman</td>
</tr>
<tr>
<td>Attitude Control Laws</td>
</tr>
<tr>
<td>Any code that involves working with</td>
</tr>
<tr>
<td>any peripheral non-processor systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 7.2: Microsatellite CVF Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use as Hardware-in-the-loop</td>
</tr>
<tr>
<td>ACS Processor</td>
</tr>
<tr>
<td>HK Processor - TT&amp;C, radios</td>
</tr>
<tr>
<td>Science Processor</td>
</tr>
<tr>
<td>Star Tracker Processor</td>
</tr>
<tr>
<td>Reaction Wheels*</td>
</tr>
<tr>
<td>Rate Sensors*</td>
</tr>
</tbody>
</table>

* Depends on availability of air-bearing table

By having these lists of what work should be done using the simulator, early flight code development for the microsatellite should prove to be efficient. Beyond that, a law of diminishing returns comes into play and work efficiency decreases. At that point, flight code development should wait until the hardware is available. The savings in time that will result by maximizing work efficiency are invaluable for a small satellite project with a short development schedule.
8. References


Appendix A: RT-Lab Simulator Components

Part I: Simulator Computers

Both Host and Target Computers

- Intel Pentium II 400 MHz CPU, 512 K L2 Cache
- GB Fujitsu Hard Disk
- MB Panasonic Floppy Drive
- 40X Toshiba CD-ROM Drive
- Motherboard ASUS P2B98-F
- Intel 740 Graphic Card 4 MB AGP
- ATX Mid Tower Casing with 235 W Power Supply

Host Computer Specific

- 17-inch Hansol 701A Colour Monitor
- 96 MB SDRAM PC-100
- Windows 95 Operating System (OS)

Target Computer Specific

- 32 MB SDRAM PC-100
- QNX OS
Part II: QNX OS Description

From QNX Webpage:

"[The] QNX Microkernel is truly a kernel. First of all, like the kernel of a realtime executive, the QNX Microkernel is very small. Secondly, it's dedicated to only two essential functions:

- **message passing** - the Microkernel handles the routing of all messages among all processes throughout the entire system
- **scheduling** - the scheduler is a part of the Microkernel and is invoked whenever a process changes state as the result of a message or interrupt

Unlike processes, the Microkernel itself is never scheduled for execution. It is entered only as the direct result of kernel calls, either from a process or from a hardware interrupt."

- kernel is very small (about 7 kilobytes of code) and fast.
- QNX system can be scaled down to 100K to fit in the ROM, or expanded to a full-featured multi-machine development environment
Features include:

- POSIX.1b clocks and timers:
  - multiple timers per process
  - timers specified in nanosecond resolution
  - flexible timer control: timers can be synchronous or asynchronous; one-shot or repetitive
- fully nested interrupts
- dynamically attachable and removable interrupt handlers
- flexible primitives for shared memory
- built-in debug primitives for local and remote debugging from anywhere on the network
- user-configurable system limits and resources
- network-wide process-naming capability
- POSIX.1b realtime draft standard process scheduling:
  - 32 priority levels
  - preemptive, prioritized context switching
  - choice of scheduling algorithms: FIFO, round robin, adaptive; all selectable per process
  - servers can have their priority driven by the messages they receive from clients
  - fully preemptive message passing

Latency

![Diagram showing interrupt processing time](image)

- Pentium 166: $T_{ih} = 3.3 \mu s$, $T_{st} = 4.7 \mu s$
Part III: Target Computer Hardware-in-the-loop Interfaces

Standard PC Serial Port

- DB-9 connector
- RS-232 serial format
- Asynchronous, half-duplex communication

Sensoray Model 626 PCI Multifunction I/O Board

- PCI bus, 32-bit, 33 MHz
- 48 digital I/O channels, TTL/CMOS compatible, each channel can be either input or output
- 20 of the digital I/O channels have edge detection and interrupt capability
- Six 24 bit up/down encoders
- 16 differential analog inputs (16 bit resolution), ±10 V range, approx. 20 μs conversion time
- 4 analog outputs (13 bit resolution) with remote sense inputs to compensate for any external output resistance, ±10 V range, approx. 200 μs conversion time
- digital and analog I/O ports each use 50 pin connectors with industry standard pinouts

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
<th>User Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIO24</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DIO23</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DIO22</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>DIO21</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DIO20</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>DIO19</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>DIO18</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>DIO17</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>DIO16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>DIO15</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>DIO14</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>DIO13</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>DIO12</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>DIO11</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>DIO10</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>DIO9</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>DIO8</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>DIO7</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>DIO6</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>DIO5</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>DIO4</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>DIO3</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>DIO2</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>DIO1</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>5V</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
<th>User Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIO48</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DIO47</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DIO46</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>DIO45</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DIO44</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>DIO43</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>DIO42</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>DIO41</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>DIO40</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>DIO39</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>DIO38</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>DIO37</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>DIO36</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>DIO35</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>DIO34</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>DIO33</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>DIO32</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>DIO31</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>DIO30</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>DIO29</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>DIO28</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>DIO27</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>DIO26</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>DIO25</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>5V</td>
<td></td>
</tr>
</tbody>
</table>

All even pins: Ground
Greenspring ATC-40 ISA IP Carrier Board

- supports four IndustryPacks (IP) modules
- 16-bit AT slot
- seven LEDs for function monitoring
- base address set with eight position DIP switch (set to 0xD0000 for this simulator)
- takes 16 Kbytes in the host address space
- up to 200 I/O lines supported in one slot
- IP-500 and IP-501 mounted on ATC-40 in this simulator

Series IP-500 Industrial I/O Pack

- 4 RS-232 communication ports
- 16-character FIFO buffers
- programmable baud rate, parity, stop bits (programmed via SystemBuild block)
- asynchronous, half-duplex communication
- all 4 ports accessed using one 50 pin connector
Series IP-501 Industrial I/O Pack

- 4 RS-422 communication ports
- 16-character FIFO buffers
- programmable baud rate, parity, stop bits (programmed via SystemBuild block)
- asynchronous, full-duplex communication
- all 4 ports accessed using one 50 pin connector

<table>
<thead>
<tr>
<th>Pin Description</th>
<th>Number</th>
<th>Pin Description</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMON</td>
<td>1</td>
<td>COMMON</td>
<td>28</td>
</tr>
<tr>
<td>RTA</td>
<td>2</td>
<td>RT C</td>
<td>27</td>
</tr>
<tr>
<td>DTR A</td>
<td>3</td>
<td>DTR C</td>
<td>26</td>
</tr>
<tr>
<td>CTS A</td>
<td>4</td>
<td>CTS C</td>
<td>29</td>
</tr>
<tr>
<td>TXD A</td>
<td>5</td>
<td>TXD C</td>
<td>30</td>
</tr>
<tr>
<td>RTS A</td>
<td>6</td>
<td>RTS C</td>
<td>31</td>
</tr>
<tr>
<td>RXD A</td>
<td>7</td>
<td>RXD C</td>
<td>32</td>
</tr>
<tr>
<td>DSR A</td>
<td>8</td>
<td>DSR C</td>
<td>33</td>
</tr>
<tr>
<td>DCD A</td>
<td>9</td>
<td>DCD C</td>
<td>34</td>
</tr>
<tr>
<td>COMMON</td>
<td>10</td>
<td>COMMON</td>
<td>35</td>
</tr>
<tr>
<td>R1 B</td>
<td>11</td>
<td>R1 D</td>
<td>36</td>
</tr>
<tr>
<td>DTR B</td>
<td>12</td>
<td>DTR D</td>
<td>37</td>
</tr>
<tr>
<td>CTS B</td>
<td>13</td>
<td>CTS D</td>
<td>38</td>
</tr>
<tr>
<td>TXD B</td>
<td>14</td>
<td>TXD D</td>
<td>39</td>
</tr>
<tr>
<td>RTS B</td>
<td>15</td>
<td>RTS D</td>
<td>40</td>
</tr>
<tr>
<td>RXD B</td>
<td>16</td>
<td>RXD D</td>
<td>41</td>
</tr>
<tr>
<td>DSR B</td>
<td>17</td>
<td>DSR D</td>
<td>42</td>
</tr>
<tr>
<td>DCD B</td>
<td>18</td>
<td>DCD D</td>
<td>43</td>
</tr>
<tr>
<td>COMMON</td>
<td>19</td>
<td>COMMON</td>
<td>44</td>
</tr>
<tr>
<td>COMMON</td>
<td>20</td>
<td>COMMON</td>
<td>45</td>
</tr>
<tr>
<td>COMMON</td>
<td>21</td>
<td>COMMON</td>
<td>46</td>
</tr>
<tr>
<td>COMMON</td>
<td>22</td>
<td>COMMON</td>
<td>47</td>
</tr>
<tr>
<td>COMMON</td>
<td>23</td>
<td>COMMON</td>
<td>48</td>
</tr>
<tr>
<td>COMMON</td>
<td>24</td>
<td>COMMON</td>
<td>49</td>
</tr>
<tr>
<td>COMMON</td>
<td>25</td>
<td>COMMON</td>
<td>50</td>
</tr>
</tbody>
</table>

Note: All of the 50 pin connectors can be hooked up to a screw pin interface block using a ribbon cable, which makes it simple to connect hardware-in-the-loop via the screw pins.
Appendix B: MOST System Data

Hardware

Magnetometer

- Manufacturer: Billingsley
- Model: TFM100G2
- No. Outputs: 3 (all 3 are orthogonal)
- Analog Output: 25 μV/nT
- Range: 0.5 V – 4.5 V
- Zero Point: 2.5 V
- Maximum Detection Value: 80000 nT
- Maximum Magnetic Field of Earth at 785 km: ~30000 nT

Sun Sensor

- Manufacturer: AeroAstro
- Model: MSS-1-B
- Photodiode: Quad detector
- Spectral Range: 190 nm - 1000 nm
- Peak Sensitivity: 720 nm
- Field of View: 67° half-angle
- Sensor Accuracy: 1 degree over 30° half-angle

Magnetorquer

- Manufacturer: Microcosm
- Maximum Magnetic Moment: 5 A-m²
- Required Voltage for Maximum Moment: 5 V
- Nominal Resistance: 3 Ω
Reaction Wheel / Rate Sensor

- Manufacturer: Dynacon
- Model: High Precision Attitude Control (HPAC) Microwheel
- Required Voltage: 8 V – 35 V
- Required Maximum Power: ~4 W
- Speed Range: ± 9000 RPM
- Maximum Torque: 3 N-m
- Speed Control Performance: ± 0.2 RPM (above 100 RPM)
- Torque Control Performance: ± 1 mN-m
- Command Rate: 10 Hz and greater

Software

Simple Serial Protocol (SSP)

SSP is an open source serial packet protocol that can be used on a multi-drop, single-master, asynchronous serial bus. The packets use SLIP (RFC 1055) framing (aka. KISS). Each packet begins and ends with a FEND (0xc0) byte. If FEND appears in the SSP packet, it is changed within the frame to FESC TFEND (0xdb 0xdc). If FESC appears in the SSP packet, it is changed within the frame to FESC TFESC (0xdb 0xdd).

The basic SSP packet format, before framing is added, is:

```
dest src type ...data... crc0 crc1
```

Each node on the bus is assigned a byte ID. If a node receives a packet with a destination byte that is not its own, it simply passes it on along the bus unchanged. The type byte indicates the functionality of the packet and what data, if any, is found within the packet. The checksum is a 16-bit CRC sent least significant byte first.
Appendix C: Simulation Block Diagrams

Part I: MOST ACS Flight Code Simulation
Sensor Data

Magnetometers
\[ \begin{align*}
\Theta_1 &= 0.000000000000 \\
\Theta_2 &= 0.000000000000 \\
\Theta_3 &= 0.000000000000 \\
\end{align*} \]

Rate Sensors
\[ \begin{align*}
\dot{\Theta}_1 &= 0.000 \\
\dot{\Theta}_2 &= 0.000 \\
\dot{\Theta}_3 &= 0.000 \\
\end{align*} \]

Sun Direction
\[ \begin{align*}
\Theta_{\text{sun}} &= 0.00000 \\
\end{align*} \]

Environment Data

Mangetorquer Torques
\[ \begin{align*}
\Theta_{\text{torque}} &= 0.00000000000000 \\
\Theta_{\text{torque}} &= 0.00000000000000 \\
\Theta_{\text{torque}} &= 0.00000000000000 \\
\end{align*} \]

Intertial Orbital Position
\[ \begin{align*}
\Theta_{\text{orbital}} &= 0.00000 \\
\end{align*} \]

Euler Angles
\[ \begin{align*}
\Theta_{\text{euler}} &= 0.000 \\
\end{align*} \]
Part II: MOST Command Verification Simulation (Modified Blocks)
<table>
<thead>
<tr>
<th>S/C Angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interstellar Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Magnetic Torques</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RW Torques</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Environment Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R/W Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sun Sensor (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Telemetry Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Impulse Ouputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decree Subblock</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>