Implementation of a Real-Time Operating System on a Small Satellite Platform

Caitlyn M. Cooke
Colorado Space Grant Consortium, Boulder, CO, 80309

The implementation of a real-time operating system for a small satellite project has an extensive list of pros and cons. A real time operating system’s multithreaded structure allows for the ability to design a complex software system with more flight capabilities and configuration options to support the success of mission goals. The increase in versatility and flexibility comes with the price of increased risk of critical errors, such as memory leaks, scheduling latency, and message queue concurrency. The following discussion will explore the real-time software architecture of the Drag and Atmospheric Neutral Density Explorer, and the benefits and major issues experienced during the final integration and testing of this system.

Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Accelerometer System</td>
</tr>
<tr>
<td>ADC</td>
<td>Attitude Determination and Control</td>
</tr>
<tr>
<td>AVR</td>
<td>Atmel 8-bit RISC Microcontroller</td>
</tr>
<tr>
<td>CDH</td>
<td>Command and Data Handling</td>
</tr>
<tr>
<td>COM</td>
<td>Communications System</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DANDE</td>
<td>Drag and Atmospheric Neutral Density Explorer</td>
</tr>
<tr>
<td>DITL</td>
<td>Day in the Life Test</td>
</tr>
<tr>
<td>EPS</td>
<td>Electrical Power System</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>HCI</td>
<td>Horizon Crossing Indicators</td>
</tr>
<tr>
<td>I2C</td>
<td>Two-wire Interface Communication Bus</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>MMC</td>
<td>Multimedia Card</td>
</tr>
<tr>
<td>NMS</td>
<td>Neutral Mass Spectrometer</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>RTC</td>
<td>Real-Time Clock</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SD</td>
<td>Secure Digital Card</td>
</tr>
<tr>
<td>SEP</td>
<td>Separations System</td>
</tr>
<tr>
<td>THM</td>
<td>Thermal System</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>

I. Introduction

The Drag and Atmospheric Neutral Density Explorer (DANDE) started in 2007 as a student developed NanoSatellite project at the Colorado Space Grant Consortium. DANDE’s mission is to study the effects of space weather and atmospheric drag on a small satellite platform. This involves coupling the data produced by a neutral mass spectrometer and a 6-axis accelerometer system to correlate the effects of neutral particles in the thermosphere to the motion of the satellite through space. DANDE consists of eight electrical subsystems, namely the neutral mass spectrometer (NMS), the accelerometer system (ACC), command and data handling (CDH), electrical power systems (EPS), thermal systems (THM), attitude control systems (ADC), communications (COM), and the separation mechanisms (SEP). The command and data handling systems is implemented as a real-time operating system. A real-time operating system uses advanced task scheduling techniques and a preemptive kernel, which allows multi-threading of processes to occur. It depends very heavily on timing accuracy of task completion rather than serial functionality, or interrupt techniques, like some simpler operating systems. This functionality allows for a

---

1DANDE Software Systems Lead, Colorado Space Grant Consortium, Boulder, CO, 80309
greater range of abilities and versatility of the software implemented on the system. The software system’s communication “bus” allows for data collection and transfer of information to all subsystems on the satellite, using an array of multi-threaded processes. A close-loop algorithm embedded in the system processes controls the attitude of the satellite, which is critical to accurate science data collection. Data processing also occurs on DANDE’s Linux avr32 processor, which couples with a Zmodem transfer system to send the processed data to the ground. The operating system also has an extensive system status checking procedure that allows for automated correction of critical errors that may occur on flight.

Though this system architecture advances the capabilities and versatility of the system, the complexity heightens the occurrence of critical system errors. After extensive system logging and debugging tools were implemented, critical mission threatening errors were revealed, such as memory leaks in system process, and concurrency in system messaging architecture. With the extensively limited memory and processing power that exists on a small satellite platform, the frequency and volume of system status data proved to be overwhelming, even excluding the large amount of science data the is produced to achieve mission success. Problems with simple Unix commands, such as directory listing, even became a source of critical stability errors on the system. DANDE’s complex real-time operating system has presented the team with quite the integration challenge.

DANDE’s real-time operating structure is unique for a small-satellite platform. In a large amount of CubeSats and NanoSats, a simpler operating system can be implemented to accomplish mission goals. Common types of operating system structures are a simple control loop, or an interrupt based control system. Such systems comprise the firmware structure of the individual subsystems that are all controlled by CDH. For example, the thermal and accelerometer systems could be considered to be simple control loops. They have one distinct task of data collection and reporting that runs in a continuous non-interrupted loop, which also call a series of subroutines in some cases. The process is completely serial and involves no educated scheduling. The EPS system firmware operators on the more complicated interrupt controlled system. The firmware has one main continuous loop that accomplishes a serial set of tasks, but also contains the capability of interrupting that set of serial tasks with a set of defined higher priority events. Such events include CDH watchdog resets, or emergency overvoltage handling. After these high priority tasks are completed, the serial loop will resume until the next interrupt is necessary. These systems have the capability of accomplishing a wide variety of tasks, but do have not the level of versatility in which CDH system operates. With multi-threading and preemptive multitasking characteristics, the CDH system is capable of accomplishing a large amount of simultaneous tasks, which the simpler operating system may not be able to easily handle.

The use of this style has an extensive series of pros and cons that make the implementation of an RTOS a difficult engineering decision to make. The capabilities of this system are vast, providing large amount of flexibility and modularity in the system architecture that others would not be able to provide. This functionality of course comes at a cost. It is arguable that this implementation producing a large amount of unnecessary overhead, suggesting that DANDE’s mission objectives could have still been achieved without the use of a RTOS. The following discussions describe the experiences of integration and final testing of the DANDE system architecture, offering insight into the challenges and rewards of a RTOS on a small satellite platform.

II. DANDE’s Command and Data Handling System Structure

A. Hardware Specifications

The CDH hardware is comprised of an Atmel NGW-100 mkII, which is the only commercially purchased board that the system flies. This board uses an AVR32 Digital Signal Processor CPU with a Linux operating system already installed. It has two Ethernet ports, and SD card and MMC card reader, a 8Mb flash memory, and USB and JTAG communication interfaces. DANDE engineers have selected a 512Mb SD card for data storage on flight. The SD card also houses non-essential binary processes, while mission critical binaries are contained on the onboard flash for increased precautions against data corruption. Below is an image of the board layout and it’s capabilities:
The kernel that runs on DANDE is a Linux version 2.6.27.6.atmel.1, which is one of Linux’s first versions of mainstream microcontroller compatible kernels, referred to as the uCLinux branch\(^3\). Running a Linux system on an embedded system was possible prior to this, but required a separate memory management unit to support multitasking, making these systems expensive and complicated\(^3\). This release allowed for enhanced real-time capabilities and a larger memory in cheaper and simple embedded systems.

DANDE’s nominal operating mode, referred to as “standby” mode, runs 9 system processes, with 1-22 threads (average about 5) each, 2 user shell scripts, and a number of system diagnostic processes, such as the kernel message logger. All of these processes and threads are performing a system or mission critical task simultaneously, putting a high load on the system. This multithreaded load needs sophisticated task scheduling capabilities with high response times in order to accomplish the goals of the DANDE software effectively. The version 2.6 kernel running on DANDE was developed to handle scenarios like the one described above with a preemptive kernel structure and improved scheduling algorithms. In prior versions, the process scheduler was forbidden to interrupt the process that is executing currently to make way for a higher priority task\(^2\). The old system would also choose the next task to be completed by calculating a score of the tasks importance\(^2\). This calculation procedure took an excessive amount of time, making complex scheduling execution time a function of the number of tasks. The 2.6 kernel allows the scheduler to stop a currently running program to allow a higher priority task to run, eliminating unnecessary delays as the other processes completes\(^2\). The new scheduling algorithm uses a simple queue FIFO structure that allocates a fixed amount of time that that task is allowed to consuming CPU\(^2\). After this time expires, the task is moved to an “expired” queue, where it is sorted based on its priority\(^2\). When all of the tasks in the current queue are executed, it switches the expired queue with the current queue and continues grabbing tasks from an already sorted queue\(^2\). This makes the timing of the scheduling algorithm a constant time algorithm, not dependent on the number of processes.

Revisions to the mutex thread blocking system were also made in this new kernel version that allows the mutex blocker to only block traffic when it is necessary, which saves even more time\(^2\). This structure is critical for the multithreaded tasking that is performed on DANDE. Below is a graph of a study performed using LynuxWorks real-time tests that demonstrates the improved performance capabilities of this scheduling system\(^3\):

---

Space Grant Undergraduate Research Symposium
B. Software Specifications:

The enhancements in multithreaded capabilities and speed of execution allow for a more complicated software system, with more on-board processing capabilities. The processes themselves are also modeled as a real time system to handle multiple tasks and information sharing across the DANDE system, as will be described below. DANDE has 13 user processes that have been written to perform mission critical tasks. It contains an abbreviated list of about 50 Linux busybox commands and system diagnostic logging tools. It also has 10 operational modes, all of which perform run at least 9 of these processes at one time. The other 4 processes only run during specific mode operations, such as science collection, or attitude control.

1. Busmessenger and the Message Queue Handler

The busmessenger process is in control of all of the bus communication between the processes on the CDH operating system and the individual subsystem AVRs (including EPS, COM, ADC, NMS, ACC, and THM). All CDH processes are dependant upon this structure from task scheduling, information requests, and information accuracy. This process runs in all of DANDE’s operational modes. This process schedules system tasks using a message queue system in both handling the information inquiry messages from the CDH processes, and retrieval of information from the subsystems. Information is funneled from all of DANDE’s processes through the busmessaging queue in a standard FIFO queue structure to handling the scheduling of message traffic between CDH and the external subsystems. Most processes use blocking communication methods for increased assurance that all messages will be communicated. Though this increases security of message accuracy, latency may occur if the message queue system is overloaded.

2. Datacollection

This process is responsible for the collection and storage of data from each individual subsystem AVR. There are three distinct data collection processes that each handles a separate task on the satellite. The clock datacollector interacts with the RTC on the COM system to keep accurate time for data stamping on the satellite. This process asks for a clock signal from the COM system on a regular interval and broadcasts the updated time every 5 minutes to the other data collectors. The battery data collector collects and stores data from the battery thermal sensors to thermal compensate the low voltage cutoff threshold for the battery charging system, and returns this value back to EPS. The general data collector collects and stores all other diagnostic data and ACC science data from the remaining subsystems. Together, these processes account for the majority of the system multi-tasking threads. They run 21 threads for each data buffer that resides on the system, plus another 13 threads for internal monitoring and logic. All data requests and information handling is funneled through the busmessenger scheduling queue structure for accurate and deterministic data packet scheduling.

3. NMS Datacollection and Commanding

The NMS data collector has the same functionality as the previously described data collectors, plus commanding capabilities for the NMS subsystem. This process collects science data from the NMS subsystem and stores it to downloadable files. It acts like a state machine to keep track of the NMS science instruments current state and
commands it successfully to the next state. The sensitivity of this instrument caused the necessity for a very sophisticated monitoring and reporting system that allows the ground personnel to know if an error occurred during operation. This process has the ability to monitor these errors and provide instant corrective action to safely recover the instrument. This data collector operates using one collection thread for all 31 data buffers and an additional four threads for internal monitoring and logic.

4. Data Processing

DANDE’s science mission and operating complexity results in the accumulation of a very large amount of data. With subsystems sampling at an average of 1Hz, the system nominally generates about 26 MB worth of diagnostic and science raw data. While the system is designed to be able to hold this large amount of data, our communication bandwidth and contact time is not substantial enough to support the downlink of all this information. The data processor compresses these raw data files and applies algorithms such as min/max/average to reduce the number of bytes of data that need to be down linked from the satellite.

5. Process and Subsystem Watchdogs

DANDE’s system is equipped with two software watchdogs that scan system activity on a regular interval to ensure nominal behavior. The process watchdog is configured to make sure the correct user processes are running in each mode. If a process dies, or does not seem to be responding properly, the process watchdog will restart that process in hope that it will recover functionality. The second watchdog, housekeeper, monitors parameters on external subsystems. If any anomalies occur on flight (i.e. A bit flip from radiation exposure), housekeeper will reset the subsystem parameter and restore nominal functionality. The process watchdog checks process activity on intervals of 10-100 seconds, while housekeeper checks subsystem parameters every 10 seconds to 30 minutes.

6. Attitude Control

DANDE has two attitude control processes that run in one specific operational mode. The torque rod command scheduling processes takes a user defined sequence of torque rod firing commands that are calculated on the ground from data collected from the HCI instruments to provide accurate alignment. The attitude determination and control manager process is a closed loop algorithm that collects data from the magnetometer directly and uses it to calculate the rate at which DANDE is spinning. The algorithm generates and executes a series of torque rod actuation commands that automatically correct DANDE’s spin rate and damping factor, which is critical to science data accuracy. This algorithm is very complicated in that it can make decisions to control DANDE’s attitude state without any human input.

7. Mode Scheduling

This process is a very simple mode managing system that takes a user defined mode selection input and informs the process watchdog to modify the process that are running. It is also responsible for running the bash scripts that control which subsystems are on or off. DANDE also has a mode scheduling process that allows the ground personnel to define a series of queued mode change commands on a specified time interval.

8. Com-CDH

This process is the communication between the ground and the spacecraft on flight. It controls the functionality of logging into the spacecraft from the InControl system used to operate and command the system. It handles the Zmodem transfer capabilities of downloading data and sending configuration files up to the satellite. It also is responsible for sending out beacon packets that contain a small amount of system diagnostic data that can be received and observed on the ground.

III. Benefits of an RTOS

The core benefit of real-time operating systems lies in how quickly and predictably it can respond to a large number of tasks. This is the more commonly used structure on large and complex spacecraft projects, such as Hubble. With the preemptive kernel structure, as implemented in DANDE, tasks are called specifically when needed, rather than in a serial loop like in a control loop based embedded operating system. This allows for better program flow and event response for more time critical tasks. The speed of the handling execution of task scheduling allows for multiple threaded applications to be run seemingly at once. This way, DANDE can be collecting data, housekeeping subsystem parameters, processing data, downlinking data to the ground and controlling attitude at one time with one shared resource. With this increased speed, the complexity of processes can
increase, allowing DANDE to use a closed loop attitude control algorithm for faster reaction to spin rate perturbations. The capability of data processing allows for a larger amount of data to be sent over the communication link, as files are put into a more compressed format on the spacecraft. The sensitivity of the Neutral Mass Spectrometer is also of great concern to the success of our mission. Due to the versatility of the RTOS platform, the NMS data collection and commanding software could be design to constantly monitor the instrument and immediately act if something goes wrong.

This functionality allows for a greater flexibility. If modifications to the system architecture need to be made, such as the logging mechanisms that needed to be implemented, major changes like this are easier to implement quickly. DANDE’s architecture is also centered mainly on system configurability. All processes on the spacecraft are controlled and modifiable externally by the process configuration file. This file is contained in an xml format, which is small enough to be easily sent over DANDE’s COM link. This means if the mission goals change, or more functionality needs to be added mid-flight, DANDE has the capability of running an entirely new set of operational modes if necessary. For debugging purposes, the configurability allows the verbose of error reporting to be changed mid-flight, the frequency of data collection, or the specification of which subsystem parameters are being monitored by the housekeeping watchdog. Problems that occur on flight will be easily debugged and resolved in this case. On a simpler control loop or interrupt system, the software that is flashed onto the system before launch is the only level of versatility that is possible on flight. For this implementation, all flight errors must be predicted and fixed prior to launch, leaving this systems no wiggle room for small implementation errors. Though DANDE’s complex architecture may be more prone to errors, the flexibility of the systems allows for on-flight debugging and modification.

IV. Hurdles of DANDE’s RTOS Structure

The previous section describes the functions performed on DANDE’s operating system and gives an overview of the amount of system processing power that is behind these operations that all must be performed for mission success. This analysis of DANDE’s task list intuitively leads to the conclusion that a fast acting, sophisticated scheduling operating system is necessary to perform this level of logic, but this also leads to a more complicated list of issues that can arise.

Integration and testing proved to be a very difficult and complicated task. During the final months of the DANDE project, the team designed and performed the “Day in the Life Test” (DITL), which ran DANDE through the phases of every mission phase that would be performed on flight. This test was designed the walk through the mission sequentially in an abbreviated time scale, encompassing the full mission in about 72 hours on continuous system run time. The first test that the team performed brought up a myriad of system critical issues that all stemmed from the operating system stability. The system could not remain stable for longer than about 3 hours. When it came to investigate the reasons for DANDE’s instability, it was discovered that our error logging and debugging system was not verbose enough to provide insight into the overall system. The only error logging that existed was problems that occurred in individual process that could cause them to fail. The ability to add the logging techniques discussed proved to be the most important tool to achieve a stable system, and is encouraged to be extensively designed into the system during the development process. The major hurdles and solutions will be discussed individually.

A. Memory Leaks and Logging Implementation

Once the system instability was observed, we realized the logging capabilities that were originally designed in the system were not sufficient for the level of detail that was necessary to reach the root of this issue. In response to this, extensive diagnostic logging mechanisms were put in place to provide more visibility into the problem. The first mechanism was a high level logging device that was written as a shell script to log system CPU usage, memory usage, load average, data file accumulation, processes active, and process error log size. This logging revealed major problems with the system architecture. We found the system memory was being fully consumed after only a few hours. This led to the implementation of an individual process logger that monitored resource consumption of each process, which essentially saved the output of the Linux “top” command to a file. From this logging mechanism, we were able to see that both the battery and general data collection processes was leaking virtual memory at over 15Kb per minute, which put a significant strain on a micro controller system such as DANDE’s. The lack of memory release form these process evoked the out of memory killer to start killing critical system processes. Once the instability of critical processes occurred, it caused the power system to reset the system.

The solutions to the problems seen here were luckily very straightforward in this case. The data collector processes were fixed to release all dynamic memory allocated for the collection process. All other system processes
were thoroughly examined to catch any other small leaks that could cause system instability over a longer period of time. We also discovered that the out of memory killer could be customized to protect certain processes from being able to be terminated when memory panic occurred. This would allow the team to debug memory issues further before the system was reset for critical failure.

B. Listing Command (ls)

The extensive memory constraints of DANDE’s RTOS were demonstrated during the debugging of the memory leak issue in Part A, but even after that was resolved, memory limitations still plagued the system. After reclaiming data collector memory, the system seemed to remain stable for upwards of 10-12 hours of DITL testing. After this amount of time, the satellite would reset, and would not regain stability unless the entire flash and SD memory was re-flashed. We found through monitoring the kernel panic log through the serial port that the out of memory killer was once again being evoked and killing the s99 initialization script. The problem also seemed to be linked directly to the runtime of the system. Through intense monitoring of the process diagnostic logger, it was found that the file listing command that was being run on startup was consuming over 20Mb of memory and causing the system to fail on every startup. We researched busy box ls and found that in order to print out the listing of files on the satellite, it stores every file name into a struct, consuming memory for temporary storage. After a consistent amount of time, the satellite would collect enough data to create enough file names to make ls’s temporary memory struct big enough to exceed DANDE’s small 32Mb memory capacity. To fix this, data collection rates were reduced significantly to lower the amount of files produced on the satellite. A custom version of ls was written to store a finite amount of files, print them, and release temporary memory in a loop, so the processes memory consumption was not a function of the number of files. Though this implementation is slower running, it allows for safe use on a low memory system.

C. Message Queue Concurrency

Once the memory consumption issues were resolved, the entire system remained stable and would no longer reset during the DITL test. Now that the system stability was under control, this allowed the team to find more critical bugs. During the next DITL test, we saw subsystems being shut off at random. After extensive EPS debugging with no results, the team began to suspect CDH as the culprit of inaccurate message being sent to turn off subsystems. Inspection of the busmessenger process error log proved this assumption to be true.

In order to monitor the flow of messages through the message queues, extensive logging tools had to be implemented to record the large volume of messages that were being sent to busmessenger and broadcasted out to the subsystems. Initial inspection of the logs lead to the conclusion that the volume of traffic was too high to even be accurately recorded. In nominal configuration, DANDE was sending over 30 messages per second through the message queues and over the I2C bus. This was placing a large amount of load on the RTOS scheduling process and the busmessenger message queue structure. Further inspection of the logs tracked the inaccurate message generation back to the command line bus messaging process, which is a shell scripting subset of bus messengers communication system. It was found that messages were being generate so frequently that the message sender identification algorithm was not generating unique sender ID numbers. This was causing message concurrency and false information was being broadcated across the bus.

The investigation of this issue identified a large number of design flaws in our current system configuration. The first fix was to make sure that all messages were stamped with a unique sender ID. A more complicated generation algorithm was developed to accomplish this. In order to improve the overall health of the system, message traffic rates were decreases in all processes. Data collection frequencies were reduced significant to reduce the amount of times the data collector queried the subsystems. This also reduced the accumulation of data product that had to be stored and download on the system, as well as decreased the CPU usage of the process. Watchdog inspection frequencies were lowered to reduce system use. Also DANDE beacon message reporting frequencies were also lowered. After the implementation these improvements, the overall health and stability of the system improved greatly.

D. Un-throttled Infinite Loops

At this point the critical systems errors were decreasing, and it was time to focus on performance. The number of processes threads running on our RTOS was causing the system load average to be undesirably high. When initially designing user processes before integration, overall system loading was not always considered. When monitoring these processes individually, or testing them on larger Linux systems, it is easy to ignore the effects that that process activities would have on the small and schedule dependent DANDE system. After putting all of these processes together, a one-minute system load average of 5-7 was exhibited during nominal standby data collection activity.
Ideally, a load average of 1 indicates a healthy system. This means that only one process is waiting to be executed at any given time through the real-time scheduler.

It was obviously to see that our current system configuration was too intensive for the capabilities of our RTOS scheduling system. With all the processes listed above trying to execute task at the same time, task queuing was becoming an issue that manifested itself in many peculiar ways throughout the system. The system exhibited occasional inaccuracy of data product, inaccurate diagnostic logging messages, and latency in command execution. This led the team to have to examine the individual process activity, and design the task execution in a more careful way to avoid overloading our system. Some processes, namely, the data collector, were found to have unthrottled infinite loops, where the process would spin through a task loop multiple times a second requesting CPU from the operating system, even if it was not needing to collect data. This activity was highly prevalent through our processes and was unfit for the sensitivity of our RTOS structure. To resolve this issue, extensive sleep functions were added to every infinite loop that occurred in process activity to make the request for processing power decrease. After this was implemented, nominal load average dropped to 2-3, moving the system closer to a nominal load.

![Load Average Before and After Loop Throttling](chart.png)

**E. The Zip Command**

After repairing the user process’s high CPU usage, there was only one process left that didn’t seem to have a clear fix for optimal performance. The data processor used the zip command to compress files into downloadable packets. Inherently, Linux busy box functions are not designed with system performance optimization in mind on a system scale such as DANDE’s. We found that the frequent activation of this command was still contributing excessively to DANDE’s poor performance. In efforts to decrease this, and still keep our packetizing functionality, the zip command was replaced with the tar command and used less frequently. From this final performance optimization, DANDE’s load average dropped to 1-2 nominal, signifying a relatively healthy system load.

**V. Conclusion**

The implementation of a RTOS on a small satellite platform comes with an extensive series of design challenges, which if not considered, can cause major problems in system functionality and stability. Integration and final testing proved to take a significantly longer time than originally anticipated. The added complexity of the RTOS produced issues that were unexpected and extremely difficult to identify. In order for this implementation to be successful, the decision to run the RTOS must be made early and designed this way from the bottom up. If the problems we had seen during integration had been fully considered during the earlier development process, a lot of these issues could have potentially been more manageable during the final phases. The importance of extensive error logging and debugging mechanisms seemed to be the most critical aspect of development. Having these techniques in place earlier could have allowed us to catch some of these issues earlier, and debugged faster during the integration phase. Some tasks performed on CDH could have been delegated to subsystems, making the volume of tasks necessary on CDH lower and allowing for the reduction of load on the system. For example, the attitude management algorithm could have been completed on the ground, similar to the system used for the torque rod alignment scheduling
process. These simplifications could have reduced the overhead seen in integration, and made the overall process of RTOS implementation more successful.

The decision to use a complex system requires an extensive amount of consideration and design, but the advantages it offers can be very beneficial if the mission calls for it. In the case of DANDE, the use of a real-time operating system allowed for a higher level of operations to take place on the spacecraft. The speed of scheduling that the real-time operating system offers makes multi-threaded tasking, as seen in DANDE’s architecture possible. The versatility of the system configuration allowed for easy design modifications, and flexible in-flight implementation techniques once the core structure of the software was established. Despite the challenges encountered, the project has succeeded with a stable and functional system that can effectively accomplish the mission goals.

References

