Flight Software Application Framework Simplifies Development for RBSP Spacecraft

W. Mark Reid and Christopher A. Monaco
Johns Hopkins University, Applied Physics Laboratory
11100 Johns Hopkins Road
Laurel, MD USA 20723-6099
Email: Mark.Reid@jhuapl.edu; Christopher.Monaco@jhuapl.edu

Abstract— **With the trend in spacecraft flight software systems toward the use of message-based architectures, flight software systems are being decomposed into several discrete applications each with a relatively narrow focus. These applications, however, share several common requirements for initialization, command processing, parameter management and telemetry generation. Even with a single common design, if each of these functions were left up to individual application developers, there would be multiple implementations. Each of these implementations would require testing and maintenance, which increases the overall development and maintenance costs and also increases the potential for bugs.**

In lieu of leaving these functions up to each individual developer of the applications the Radiation Belt Storm Probes (RBSP) Flight Software development team has isolated the commonality across all of the flight software applications and created an application framework. This framework separates the software functions that are common to all applications and the software functions that give a particular application its unique personality. An application deployment tool was also created that allows a developer to create a new application using this framework and insert it into a flight software system in a matter of minutes.

The use of an application framework and deployment tool speeds up software development by enabling the creation of an executable application that can receive commands and generate basic telemetry in minutes. This approach, through the separation of the common application code and specific application code allows all applications to use the same overall design while enabling the batch maintenance of the common functionality. This paper discusses the design of the RBSP application framework, deployment tools, the flight software maintenance model, as well as the impact on the flight software development cycle.

TABLE OF CONTENTS

1. INTRODUCTION

The RBSP mission, which is part of NASAs Living With a Star Geospace program, is being designed and built to help us better understand the suns influence on the Earth and near-Earth space by studying the planet's radiation belts on various scales of space and time. The mission consists of two RBSP spacecraft, each hosting an array of instruments that will provide the measurements needed to characterize and quantify the processes that produce relativistic ions and electrons. They will measure the properties of charged particles that comprise the Earths radiation belts and the plasma waves that interact with them, the large-scale electric fields that transport them, and the magnetic field that guides them. [1] Both spacecraft will be launched on a single EELV launch vehicle and will operate in highly elliptical orbits, spending a substantial part of the mission life in the Van Allen radiation belts. These spin stabilized spacecraft have been uniquely designed for this mission environment. [2]

Each spacecraft hosts an Integrated Electronics Module (IEM) which contains a flight computer consisting of a RAD750 Single Board Computer (SBC), a Solid State Recorder (SSR) and custom spacecraft interface hardware. Resident in this computer is the Command and Data Handling (C&DH) flight software. The C&DH software is a set of functional applications and libraries, which were implemented to be used with the core Flight Executive (cFE) software developed by NASA Goddard Space Flight Center (GSFC) and the VxWorks operating system. The cFE software provides standard services with a standard application programmer's interface (API).[3] The RBSP C&DH software is composed of seventeen (17) unique applications and eight (8) libraries. Each of these applications and libraries has been designed and implemented to perform a specific set of functions. The full list of applications is provided in Table 1. Examples of the libraries include generic libraries such as the Consultative Committee for Space Data Systems (CCSDS) File Delivery Protocol (CFDP) library, Memory Object library, EEPROM library, and mission specific libraries like the RBSP specific Spacecraft Interface (SCIF) library.

This paper is focused on the common framework provided for the complete set of RBSP flight software applications. In order to create a cohesive set of applications with common methods of processing commands and generating

1 978-1-4577-0557-1/12/$26.00 ©2012 IEEE

telemetry, the RBSP flight software team developed an application framework that allows for applications to be generated more rapidly and to be more easily integrated into the system. The remainder of this paper will discuss the goals of the application framework, the approach taken, necessary tools developed and the impact on the RBSP flight software development.

Table 1: RBSP Flight Software Applications

Commanding and Autonomous Operations Applications	
- Command Ingest (CI)	Performs CCSDS uplink protocol, including code block verification, Command Link Transfer Unit (CLTU) processing, frame level verification and COP-1 protocol.
- Command Manager (CM)	Provides macro level commanding, command prioritization, command sequencing, command status reporting and packet level verification of command formatting.
- Time-tagged Commanding (TT)	Provides onboard storage and triggering of commands or command sequences based on spacecraft time.
- Autonomy Engine (AUT)	Performs autonomous checking of telemetry and execution of stored commands or command sequences based on operator specified conditions being met onboard.
Spacecraft and Instrument Interface Management	
- Instrument Manager (IM)	Provides storage for instrument time tagged commands and manages all instrument command and telemetry interfaces. Performs verification of instrument data.
- Spacecraft Interfaces (SI)	Manages commands and telemetry from other spacecraft subsystem interfaces including the Power Distribution Unit (PDU), Power System Electronics (PSE), Transceiver and Remote I/O (RIO) units.
- Guidance and Control (GC)	Processes all sun sensor data and performs calculations of spin period, spin phase and sun angle information which is needed by the instruments onboard or for autonomous fault protection.
Data Recording and Playback Applications	
- File Manager (FM)	Manages onboard file systems and performs file system operations such as directory listing, file move and file delete operations. Also performs allocation management based on file data source.
- Recording (REC)	Records operator selected data to specified files and specified rates. Provides automatic file size limiting and re-open on full capabilities.
- Playback (PB)	Performs CCSDS File Delevery Protocol (CFDP) playback of recorded files in both acknowledged and unacknowledged mode. Performs retransmission of missed data in acknowledged mode and supports auto
- Telemetry Output (TO)	Manages the downlink hardware interface and interleaves realtime and playback data. Also supports a unique downlink virtual channel (VC) for space weather broadcast data.
Memory Management Applications	
- Memory Manager (MM)	Provides raw memory load, dump and copy capabilities necessary for on orbit software maintenance and reload of non-volatile and volatile memory.
- Memory Object Handler (MOH)	Provides load, dump and copy operations of logical memory objects such as stored command macros and configurable parameter tables.
- Memory Scrub (MS)	Supports hardware scrubbing of SBC memory and both hardware and software scrubbing of the SSR.
Software Scheduling and Monitoring Applications	
- Scheduler (SCH)	Provides 100Hz schedule for initiating software execution and collection of housekeeping telemetry data.
- CPU Monitor (CPU)	Monitors and reports CPU usage.
- Housekeeping Telemetry Monitor (HTM)	Provides a subset of spacecraft housekeeping data to a ground test port for monitoring during spacecraft Integration and Test (I&T).

2. RATIONALE AND GOALS

The goals of the application framework are actually quite common goals for software development. The overarching goals were to minimize the development effort by maximizing the common code in our code base. In order to achieve this, we identified five (5) specific goals for the application framework. Each of these goals and the actions taken to meet each goal is provide here.

Goal 1: Simplify the creation of a new cFE application for RBSP FSW development.

This goal was achieved by creating scripts and deployment tools that operate on an application template to auto generate source code for each new software application. This functionality is referred to by the RBSP team as the "Application Wizard". The Application Wizard allows a developer to auto generate a complete, ready to compile, application, which includes all necessary header files and make files. This approach enables the automation of new application creation and therefore shortens development time.

Goal 2: Separate application common code from the application specific code.

This goal was met by creating an application template that supports common software, which should not be modified by the individual application developer, as well as a framework for application specific software, which the developer fills in as the application is developed. This allows for some software to be developed once and used in all applications while providing structure for application specific software, which is yet to be developed. It also allows for later deployment of modifications to common software without requiring developers to modify their specific application code.

Goal 3: Baseline and maintain uniform application flow and data structures.

In order to meet this goal, we developed a common application structure as well as defined naming conventions for functions, commands, data structures, variables and messages across all applications. Having common naming conventions for command functions, telemetry data and common processing allows developers who may be unfamiliar with a specific application to more easily understand the software structure and flow.

Goal 4: Reduce code review material.

This was successfully accomplished by creating the application framework early in the project and reviewing it prior to developers beginning to develop their specific applications. The team reviewed the entire template application, including the common command and telemetry functions and the application specific framework that would

be used to generate the unique RBSP applications. By reviewing this code once, up front, the team did not need to review code for this functionality when reviewing all seventeen unique applications. This approach also reduces the amount of code that each developer is responsible for developing, therefore reducing the overall amount material that needs to be reviewed and tested.

Goal 5: Enable sharing and/or consolidation of unit and integration test procedures.

In addition to the common application code and the application specific framework, the application template also includes both common unit test code and a framework for application specific unit test code. This unit test code can be conditionally compiled into the software during early development and unit testing, but then later compiled out for system testing and final deployment. This provides for common unit tests to be developed just once for the portions of the applications that perform common functions. The common unit test framework also improves the ability to document and audit software development processes.

3. Design of Application Framework

When looking at the common requirements for all seventeen of the C&DH applications there were clearly common functions that must be performed by each application. These included application initialization and the processing of application commands and the collection or reporting of housekeeping telemetry. The application framework is built around three template application packages: TMP_app_main, TMP_app_specific, and TMP_cmds, as shown in Figure 1.

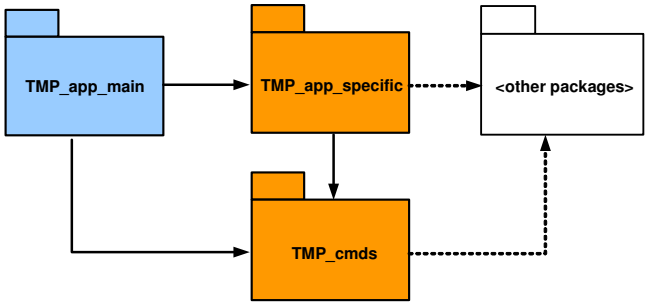

Figure 1: Application Framework Packages

The TMP_app_main package depends on TMP_app_specific and TMP_cmds packages as indicated in Figure 1 by the arrows. In turn, the TMP_app_specific and TMP_cmds packages may have additional dependencies as indicated with the dotted lines.

These three template packages are deployed through a set of scripts, which create a unique instance of the application for the developer. In this unique instance, the "TMP" in the package name, indicating a template application is replaced with the developer specified identifier. For example, the Command Manager (CM) application main package would be CM_app_main.

The out-of-the-box application, as generated from the template, is single threaded and event/message based. After completing application initialization it will enter a message-handling loop, in which it pends on its cFE Software Bus pipe and processes the received messages as they arrive. Message handling (and command handling) is performed in the context of the application's main thread using a table lookup into a message/response table. In some cases, however, developers may choose to handle a command by deferring it to an application specific, low-priority task that is not part of the template. This is to avoid occupying the application's primary thread for extended duration commands.

The application framework was designed to include the following features:

Feature 1: Message/Action Registration and Handling

One of the main features of the template is that it offers APIs to attach/detach message handlers to incoming messages. The main thread of the application pends on the application's cFE Software Bus pipe, when a message is received it locates the message registration, performs size verification and dispatches the message using the registered handler function. This feature is used by the template to perform registration with the scheduler, to invoke it's periodic activity and to receive and dispatch commands into the application's command handler.

Feature 2: Generalized Command Handling

The application framework includes APIs to register application commands, providing the ability to attach a command handler to a command code. This relies on the underlying implementation of the message/action registration and handling while adding command-processing specific functionality. Framework processing of commands adds basic command verification of command registration, command length and command acknowledgement. Application command handling information including the command identifiers, lengths and command execution routines are stored in a table. This table is normally built at initialization time through the common registration routine. This mechanism allows application developers to add specific command routines to their applications without requiring modifications to other software packages. The common application software APIs to perform command registration can be invoked from the specific application software initialization code to identify the command messages to be processed by their application, and the associated lengths of these commands.

Every command issued to an RBSP flight software application is responded to with either a positive acknowledgement (ACK) indicating command execution or a negative acknowledgement (NAK) indicating a command

error. When a command is received by the application, a common TMP_ProcessAppCmd() routine performs the verification of the command and hands the execution of the command to the specific registered command handling routine. This common software performs checks to ensure that the command received is a known, registered, command and that it is the proper length. If an unregistered or improper command is detected the command will be NAK'ed immediately by the framework code, indicating a command packet error. A proper, validated, command will be passed along to the common or specific function that implements the command for further argument checking and eventual execution. The common framework also provides a TMP_CmdAckNak() routine which the specific application command execution routines can call to acknowledge the eventual execution of the commands.

In addition to this validation of all application commands, the application framework provides command execution routines to process commands that are common to all applications. These commands include a No-operation (NOOP) command, and commands to set counters, clear tell-tale telemetry points as well as commands to access registered application memory objects, such as parameters or data structures. These execution routines perform further command verification of these common commands and generate the command acknowledgement with either the ACK or NAK, as appropriate.

Feature 3: Housekeeping Telemetry Generation.

Each RBSP flight software application provides a housekeeping telemetry packet, which provides status of the application software. These packets can include counters, flags (also known as tell-tales), or other state telemetry. The application framework provides a common housekeeping packet and a TMP_GenHousekeeping() routine that populates the common application housekeeping, timestamps the housekeeping telemetry packet and sends it for further recording or real time downlink. The common application housekeeping contains an application execution cycle counter as well as command receipt, execution and failure counters. These counters are maintained and incremented as appropriate for every command that is received by the application.

Additionally, this common TMP_GenHousekeeping() routine invokes an application specific routine, TMP_CollectSpecificHK(), which is filled in by the application developer. This routine populates the application specific portion of the housekeeping packet. For example, this is where an application like the Guidance and Control (GC) would report things like spin period calculation errors, maneuver activity or the receipt of invalid sun sensor data.

Feature 4: Application Initialization

As already identified, specific command messages are registered through the application framework at initialization time. It is at initialization time that many of the connections between the application common code and the application specific code are established. In addition to the registration of commands for the common command handling, other scheduling and message registrations are performed at this time. For example, the Telemetry Output (TO) application may register a list of packets to be placed in the real time downlink stream during initialization. This registration is used to build up a table of message identifiers and the routine to call upon receipt of these packets.

In addition to command and telemetry registrations, the application framework supports registration of memory objects and event filters for events that the application may generate. The memory object registration allows the common application software to process parameter tables or other memory objects associated with a specific application by invoking the specific object handling routines provided by the application specific code. In order to support the extension of the application at initialization time, the application framework provides an application specific TMP_AppSpecificInit() routine and a similar TMP_InitWorkingCriticalData() routine, that are filled in by the developer, and are called by the common application initialization as each application is initialized.

All applications have the same startup sequence, the same approach to registration with the scheduler application and the same approach to periodic invocation. Therefore, the use of the framework for application initialization also provides the ability to perform startup synchronization where all of the applications begin to execute in a consistent fashion.

Feature 5: Stub Functions to Add Application Personality.

Several stub functions are provided by the framework. These routines are intended to be filled in by the specific application developers. As already indicated, stub functions are provided for application specific initialization and housekeeping collection. In addition to these routines, the application framework provides a stub function, TMP_DoPeriodicWork(), which is executed at the particular application's scheduled rate to perform periodic processing. For example, the GC application uses this function to perform 1Hz calculations of spin period and spin phase to be provided to the instruments onboard. Another stub function, TMP_ResetAppSpecificData(), is used by applications to return specific data to a known initialized state. This is most often to clear all counters and tell-tales associated with the application.

Feature 6: Unit Test Framework.

In order to provide a common unit test structure, two additional packages, TMP_unit_test_common and TMP_unit_test_spec are provided for each application generated from the application framework. This is shown in Figure 2.

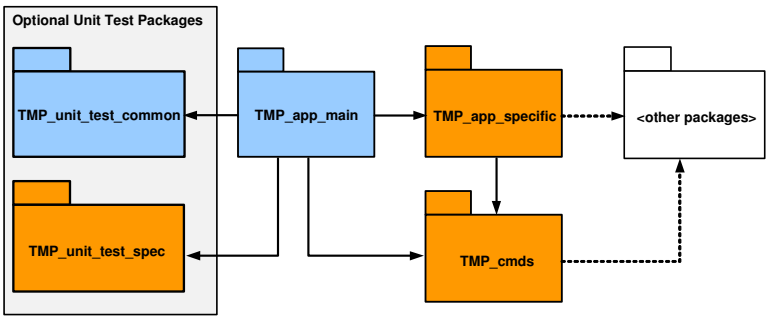

Figure 2: Optional Unit Test Packages

The TMP_unit_test_common file provides common unit test routines for testing the common application framework routines. The TMP_unit_test_spec file provides a placeholder to be filled in the application developer for routines to test the application specific code. These two files are conditionally included into the application build through the use of an optional "UNIT_TEST" flag which is used by the make file. This allows the unit test code to be compiled in during early development and developer testing, but to be left out of the build for final system tests, acceptance testing and operational deployment.

The common and specific unit test cases are enumerated and can be invoked by command. For RBSP these command definitions were in a "red-tag" component of the command database during development and have been removed for flight.

4. Deployment Tools

Using the application framework template described in the earlier sections of this paper, an application developer is able to generate a new application in the matter of just a few minutes by using an Application Wizard. The Application Wizard consists of a Perl script AppWizard.pl, which takes limited user input to define the output application. The user is able to specify the mnemonic by which that application will be referred. This mnemonic is used as a prefix in naming the application's functions, data structures, and messages during the creation of the application. Application mnemonics used for RBSP are CI, CM, TT, AUT, etc. as listed in Table 1. Additionally, the user provides options to the script such as "-new" to generate new application specific code, and "-ovr" to overwrite existing target files. The absence of the "-new" argument causes the application wizard to only generate the common application files. The output of this script consists of the following files where TMP is replaced with the provided mnemonic:

Common files generated always:
TMP_app_main.c – common application routines
TMP_app_main.h – common constants, structures, etc.
TMP_unit_test_common.h – common unit test prototypes
TMP_unit_test_common.c – common unit test routines
template_readme.txt – developer instructions

Additional specific framework files generated with "-new" option:
Makefile – make rules for compiling the application
TMP_app_specific.c – application specific routines
TMP_app_specific.h – specific constants, structures, etc.
TMP_cmds.c – application specific command routines
TMP_cmds.h – application specific command structures
TMP_msgs.h – application specific message structures
TMP_unit_test_spec.h – specific unit test prototypes
TMP_unit_test_spec.c – specific unit test routines

Upon initial execution of this AppWizard.pl script with the "-new" option, the developer can follow the instructions in the template_readme.txt file to make the necessary modifications to the build files to include the new application into the cFE build environment. The application can then be immediately made using the command line "make" and the provided Makefile. This automation allows developers to generate a new application without having to spend the time building the application infrastructure. Instead the developer can focus on the specific functionality that the application is meant to host.

5. Flight Software Maintenance Model

The use of cFE and this application framework provides a significant amount of flexibility in partitioning the flight software into small reloadable applications. Each application is independently built and can be independently loaded to the spacecraft, without requiring a complete reload of the flight software. The application framework and associated deployment tools give the team a quick and easy way to instantiate a base for any application we want to add to the system. After the short process of creating a new application, the developer can immediately focus on satisfying requirements rather than replicating application infrastructure.

Having this common application framework introduces the question of what to do if a change needs to be made to the framework itself. If a change is required in the common portion of the template application, the answer is very simple. The framework is designed to separate the application common and application specific code. The deployment tools have been developed with the option for only the common framework packages to be regenerated. As a result, a developer can deploy changes to application common template code to all of the applications in the system, in a matter of seconds or minutes. For RBSP, an update of the common software is done by making the change to the effected template file (e.g. TMP_app_main.c), and then redeploying this for all of the applications. This redeployment is done through a simple shell script that

executes the AppWizard.pl script, for each application, without the "-new" option. This generates only TMP_app_main and TMP_unit_test_common packages, but leaves the application specific software and makefile unmodified.

If, however, a change is identified that impacts the application specific framework code, there are really two options. One option is to update the template files and have developers individually make the changes to their specific applications. This will ensure that any new application would pick up the desired change, but leaves it up to the specific application developers to retrofit the change into existing applications. A second approach would be to modify the template files and regenerate each application in a new location with the "-new" option. Then a careful merge would need to be performed on each existing application to incorporate the change.

The framework, itself, contains a version number that is updated as the template is updated. The version number is available as a dumpable data structure during execution of the FSW and it can then be easily demonstrated that all applications are up to date with respect to the application template.

6. IMPACT ON THE DEVELOPMENT LIFECYCLE

The use of the application framework had a significant positive impact on the development of the RBSP flight software. By creating the template application and associated application wizard scripts, a complete set of RBSP C&DH framework applications was instantiated and deployed to the build environment prior to any application software being generated. Each functional application was then assigned to a developer to implement and unit test. The RBSP software was developed in three major builds with new applications completed in each build. In the first build, the flight software implemented the command and telemetry interfaces necessary to verify basic hardware functionality. In the second build, applications necessary for autonomous operations and development of operational Autonomy objects were implemented. In the last build, the SSR related, record, playback and file management applications were completed. This process allowed for a complete system infrastructure to be present from the earliest build and for the complete functionality of the system to be incrementally added over time.

As each individual application was developed, code walkthroughs were performed of newly developed application specific software. However, the common application framework software had already been reviewed, and was, therefore, not reviewed again. Proper code reviews require a significant amount of time and effort from multiple developers. Reviewing the application framework once for all applications proved to be quite a savings in effort. Similarly, the software implemented in the application framework was unit tested once. This saved a significant amount of redundant test generation and execution.

The amount of effort saved from using the application framework is best quantified by looking at the overall effort of the C&DH software development, and the amount of software provided in the template application. The RBSP C&DH software consists of approximately one hundred twenty two thousand executable lines of code (122K LOC). The template application consists of approximately 2K LOC., When the reuse of this code is multiplied by sixteen additional applications (1 original, 16 reused, 17 total), the total amount of functionality provided would be equivalent to individual developers developing and testing 32K LOC, or approximately one fourth of the total RBSP software.

The overall impact on the RBSP flight software development effort was a shorter development cycle, and specifically savings in code reviews and unit testing. Additionally, it added flexibility and ease in adding new functionality with each incremental builds. It also allowed for system integration much earlier in the build lifecycle by providing complete framework applications early in the development life cycle, with specific functions being added as the new functionality was needed. Additionally, using this software in all applications reduced the likelihood that an error in the application framework would go undetected. This software is currently running in multiple test development systems as well as the two RBSP spacecraft. And since the same source software is running in all applications on each of these systems, if an error is detected in one application, it can be corrected and redeployed into all applications.

A number of updates to the application framework were performed through the course of development of the RBSP flight software. Each time the deployment process to all 17 application was quick and without problems. Had this functionality been implemented 17 times, without a systematic approach to maintenance, the likelihood of errors introduced during maintenance is undoubtedly greater.

The presence of a convenient, uniform unit test framework available to all application developers and its use resulted in high quality software development process artifacts. The RBSP mission received positive feedback on the software development process and related documentation, in particular documentation of unit testing in a 2010 NASA Contract Assurance Services (NCAS) audit.

7. FUTURE WORK

Although, an attempt was made to prevent any code from being implemented in the application template that could otherwise be put into a library, a reentrant library based implementation of the message/handler registration has been developed as part of an unrelated internal IRAD and would offer an improvement to the template and reduce total code size.

Additional decoupling between the application common and application specific code could further improve the maintainability and reusability of the template. For example, using hooks and registration APIs in place of the stub functions provided by the template.

8. SUMMARY

In summary RBSP benefited from the development of an application framework, which allowed for a top down incremental build approach. The overall development effort was reduced by an estimated 25 percent when compared to having each developer create his or her entire application from scratch. This application framework reduced the total amount of testing and code reviews required and reduced the probability of having errors missed during developer testing. Having a single common design and single common code base had the advantages of improved software review, testing and maintenance, and in the end reduced the total software development effort, producing a quality C&DH flight software system.

As RBSP is the first mission developed by APL using this architecture, it is expected that future cFE application software development at APL will benefit from this application framework. The common software in each application will make it much easier to reuse this software on future missions and allow developers to more easily transition their focus between applications. This development model and template software will allow for much faster and earlier development on future missions that will use the cFE software.

REFERENCES

[1] Johns Hopkins University, Applied Physics Laboratory RBSP Web site: http://rbsp.jhuapl.edu

[2] Kirby, K.; Fretz, K.; Conde, R.; Herrmann, C; Maurer, R.; Butler, M.; Ottman, G.; Reid, M.; Srinivasan, D.; Rogers, G.; Bushman, S.; "Radiation Belt Storm Probe Spacecraft and Impact of Environment on Spacecraft Design", IEEE Aerospace Conference, Big Sky, MT, 2012

[3] Jonathan Wilmot, NASA Goddard Space Flight Center, "Implications of Responsive Space on the Flight Software Architecture", 4th Responsive Space Conference, April 2006.

BIOGRAPHIES

Christopher A. Monaco is a member of the Senior Professional Staff at the Johns Hopkins University Applied Physics Laboratory. As a real-time embedded software engineer in the Embedded Applications Group, Chris has most recently been focusing on both flight and ground software development for the RBSP mission. Chris is currently the Boot and C&DH Flight Software Lead on the NASA Solar Probe Plus mission. He was the Lead Flight Software Engineer of the C&DH and Earth Acquisition (EA) software applications on the STEREO mission. Chris has over 10 years of experience in real-time embedded flight software systems on three NASA missions. Chris has a B.S. in Physics from Clarkson University, a B.S. in Computer Science from Clarkson University and an M.S. from the Johns Hopkins University.

Mark Reid is a member of the Senior Professional Staff at the Johns Hopkins University Applied Physics Laboratory. He works in the Embedded Applications Group and has over 20 years of experience in the development of flight software for various NASA missions. His experience includes Guidance and Control, Command and Data Handling, Power Subsystems and Autonomy Subsystem software development. Mark is currently the Flight Software Lead on the Radiation Belt Storm Probes (RBSP) mission. Mark has a Bachelor of Arts (B.A.) in mathematics from Western Kentucky University and an M.S. in Computer Science from the Johns Hopkins University.