

## Implementing Centralized Command Management in a Distributed Plug-and-Play Software Architecture

Christopher Monaco

Johns Hopkins University, Applied Physics Laboratory  
11100 Johns Hopkins Road, Laurel MD 20723; 240-228-5387  
Christopher.Monaco@jhuapl.edu

W. Mark Reid

Johns Hopkins University, Applied Physics Laboratory  
11100 Johns Hopkins Road, Laurel MD 20723; 240-228-2667  
Mark.Reid@jhuapl.edu

### ABSTRACT

The Johns Hopkins University Applied Physics Laboratory (APL) developed a flight software system which has been used for multiple NASA missions. One fundamental component of this software system is a centralized command executive that has been successfully demonstrated on four ongoing missions: Thermosphere, Ionosphere, Mesosphere, Energetics and Dynamics (TIMED), MERCURY Surface Space ENVIRONMENT GEOchemistry and Ranging (MESSENGER), New Horizons, and most recently Solar TERrestrial Relations Observatory (STEREO). Through the development and operations of these missions, developers and operations personnel gained a significant amount of experience with this flight software system. This system provides all of the command capabilities required to meet the common requirements of these missions. In order to support mission requirements of long round trip light time or long periods between command contacts, all of these spacecraft perform normal operations through onboard command sequences and incorporate autonomous fault protection capabilities. This paradigm introduces requirements to provide some degree of command prioritization and the need to guarantee command execution order. The centralized command executive meets these requirements.

Plug-and-play architectures break this paradigm in that they often require each component to be capable of handling its own commands. In order to truly distribute functionality into plug-in modules or applications, each module or application needs to be self sufficient. This paper details efforts to prototype a centralized command management application to be used within a plug-and-play flight software architecture. This prototype was created for evaluation for the Radiation Belt Storm Probes (RBSP) mission and implements much of the functionality of the existing STEREO command executive flight software within a plug-and-play messaging architecture. Within this prototype application, the familiar aspects of command priorities are maintained, and command execution order is guaranteed where required. This paper discusses the performance of this prototype as well as approaches to reduce the overhead required to support centralized command management, while still providing for distributed command execution.

### Table of Contents

Introduction .....	1
STEREO Heritage Command Executive .....	2
Plug-and-Play Concepts and cFE .....	4
RBSP Prototype Command Manager .....	5
Evaluation and Performance .....	7
Conclusions .....	8
Acknowledgements .....	8
References .....	8
Biographies .....	8

### INTRODUCTION

The Embedded Applications Group within the Space Department at APL is responsible for the development, test and operational support of flight software for deep space and near earth missions. The most recent of these missions was STEREO, launched in October 2006. STEREO is a two year mission which employs two nearly identical observatories to provide the first ever 3-D images of the sun. More detailed information about the mission can be found in the STEREO Mission Guide.<sup>1</sup> As a result of these missions, the Embedded Applications Group has developed a core set of heritage software that meets the operational needs of the various missions. One area of this software that has been

strongly influenced by both the Mission Operations (MOPS) team and the spacecraft Integration and Test (I&T) team is Commanding. How commands are managed, processed and ultimately executed is a direct result of the mission's concept of operations and the requirements necessary to support testing.

In an effort to increase software quality and reuse, improve the flexibility of our software products and also maintain some of the core strengths of our existing system, the group has undertaken activities to look at alternate architectures. One part of this effort, which was initiated in support of RBSP, was to evaluate a more distributed, data driven, architecture that demonstrates many of the key elements of plug-and-play. As part of an architecture selection trade study performed for RBSP, a small team prototyped heritage software components with the core Flight Executive (cFE), a software product developed by the National Aeronautics and Space Administration (NASA) at the Goddard Space Flight Center (GSFC). The cFE provides a layered, message based, software architecture that supports plug-and-play concepts.

This paper presents a high level overview of the STEREO command software and how it provides all of the functionality required to support MOPS and I&T. It also gives a brief overview of the Plug-and-Play concepts addressed by cFE and how they may seem counter to these requirements. Next it details the prototype software developed for RBSP and identifies

areas where software performance can be improved. Finally this paper provides recommendations and conclusions about the further development of this prototype application and for eventual flight use.

### STEREO HERITAGE COMMAND EXECUTIVE

The STEREO Command and Data Handling (C&DH) system design draws heavily upon the APL heritage architecture. It is comprised of several concurrently executing tasks on a single processor all running within a single application. The STEREO spacecrafts employ a centralized command system. All commands execute within the context of a single task referred to as the "Command Execute Task", while specific command functionality is implemented through public interfaces to the various C&DH software packages which all conform to a specific standard for command execution and command checking routines.

As shown in Figure 1, the STEREO command system is composed of a Command Check task, Command Execute task, interface data structures, and stored command sequences which will be subsequently referred to as macros. STEREO implements several types of commands including Real-time commands, Time Tagged commands and Autonomy commands. Real-time commands are received from the Mission Operations Center (MOC) and are executed immediately while Time Tagged commands may be loaded by any of the various command sources but are

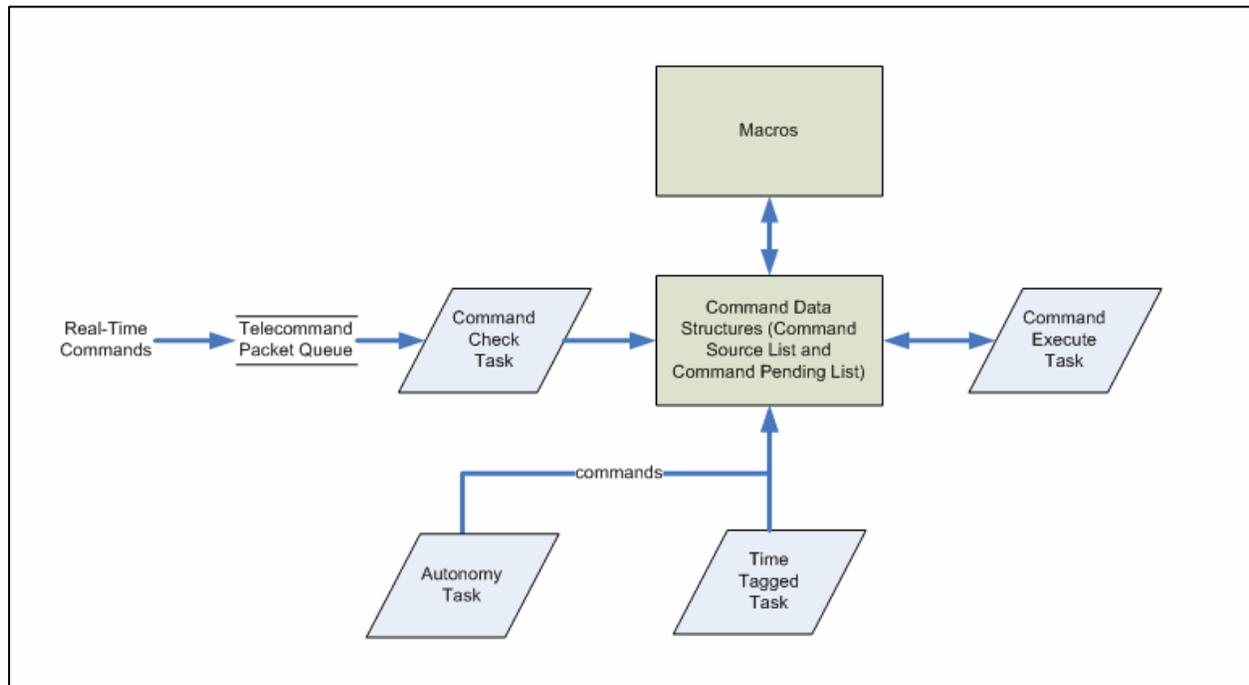


Figure 1 - STEREO High Level Command System

stored onboard for execution at a later time. Autonomy commands are those whose source is the Autonomy subsystem of the C&DH application. These commands are issued when logical expressions referred to as Autonomy Rules evaluate to TRUE. The Autonomy subsystem is responsible for autonomous operations, fault protection and spacecraft health and safety. Commands from each of these sources are inserted directly into command data structures by the Command Check, Time Tagged or Autonomy task for execution by the Command Execute task as illustrated in Figure 1. The Command Execute task will then identify each command by Op-code and will call the appropriate public routine within the associated software package. One critical point with this design is that all commands execute within the context of a single task and therefore execution order and priority is guaranteed. Additionally, the system provides a macro capability to implement stored command sequences on STEREO. Macros may be referred to from any of the sources: Real-time, Autonomy or Time Tagged. Commands from macros are similarly placed into the command data structures for execution by the Command Execute task. Autonomy and Time Tagged subsystems depend heavily upon the existence of macros.

The STEREO command system must be able to orchestrate the execution of each of these command types concurrently while meeting the requirements of Real-time commands, Autonomy commands and Time-tagged commands. In order to allow the different command types to be executed concurrently, the STEREO command system allows command execution to be prioritized. This is primarily facilitated by the command data structures. Command prioritization is particularly important in the relationship between Real-time commands, Real-time macros and commands originating internally from the C&DH system. Real-time commands are assigned the highest priority level and can therefore be used to abort any active macros that may have been initiated by the Autonomy or Time tagged command tasks. Real-time macros are the next highest priority, while the remaining priority levels are available for commands originating from within the C&DH system. Commands are provided to suspend and resume commanding capabilities at various priorities and to abort commands or macros executing at a lower priority. There are also commands to abort specific Autonomy Rule or Time-Tagged commands including any macros that have been started as a result of these commands. Individual macros cannot be aborted explicitly; however, they may be indirectly aborted by either aborting the Autonomy Rule or Time Tagged command that triggered its execution or by aborting the command priority in which the macro is executing.

The STEREO command check subsystem is responsible for validating Real-time commands as they are received prior to execution. Like the specific command execution functionality, the functionality that implements the command checking for each command is implemented within the relevant software package and according to a standard interface. For example, a command to set the downlink rate has a command check routine as well as a command execute routine. The command check routine is called by the Command Check task in order to validate the command is properly structured and contains valid arguments at uplink time. Although the Command Check task only operates on Real-time commands, the specific routines that implement the individual command checks are also utilized to validate macros, Autonomy Rule commands, and Time-tagged commands when they are loaded to RAM. At execution time, the specific command execute routine is called by the Command Execute task and performs any additional verification of the command, such as determining if the command is valid within the current spacecraft mode. A point to note here is that any command execution error in a macro causes the macro to be aborted.

The interface between the Command Check task, Command Execute task, Autonomy task, Time Tagged task and the macros is a pair of command data structures: The Command Source List and the Command Pending List. These two data structures organize and prioritize the commands pending execution. For each command priority at which there are pending commands, there is a linked list of all of the commands at that priority. This set of linked lists is collectively called the "Command Pending List". The Command Source List is a linked list of all of the unique priority levels for which commands are pending, and includes pointers to the first command at that priority in the command pending list. The command source list also maintains a macro execution stack that allows macros to call other macros and still return to the parent macro to execute subsequent commands.

Since all commands are funneled through the single Command Pending List, the execution sequence of commands within each command priority level is identical to the "arrival" sequence of the commands at that priority. Within a priority the commands may come from any source that may generate commands at that priority: Real-time commands may be at priority 0; Real-time macros, priority 1; Autonomy commands 2-15; and Time-tagged commands 3-15. The execution order of commands across priorities may be altered through the use of additional "SLEEP" and "PAUSE" commands. A "SLEEP" command suspends the execution of commands at the priority at which it is

executed for a specified duration, while allowing lower priority commands to execute. This may be useful for onboard operational sequences that contain long duration activities such as warming cat-bed heaters. A “PAUSE” command suspends the execution of commands at the priority at which it is executed for a specified duration, and also suspends all lower priority commands. This is useful for inserting delays into high priority fault protection sequences which must execute from beginning to end without interruption.

In addition to executing the commands the Command Execute task maintains an onboard command history by logging the command execution status. This command history is transmitted to the ground as part of the normal real time spacecraft housekeeping telemetry, as well as in the event and anomaly reporting. It is also recorded onboard during periods of non-contact since it also contains the history of all autonomous or time-tagged commands.

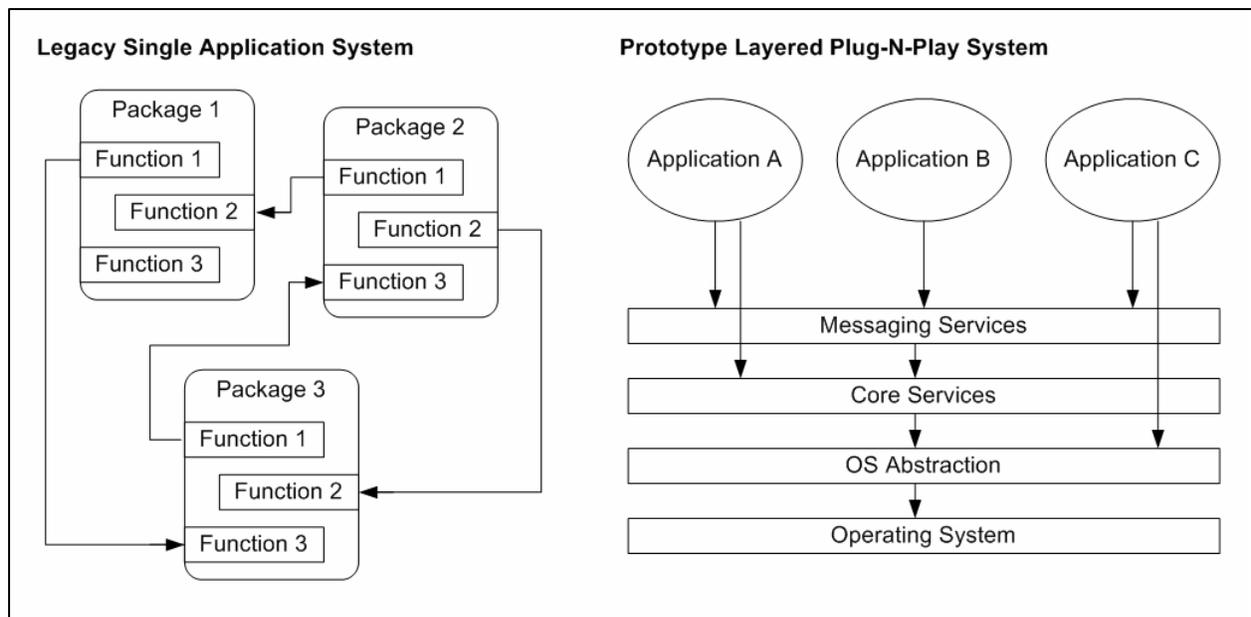
**PLUG-AND-PLAY CONCEPTS AND CFE**

As part of an architecture selection trade performed for RBSP, APL created prototype applications based on heritage software and performed an evaluation of the use of this software with NASA’s cFE. The cFE provides a layered architecture which consists of an OS abstraction layer, hardware abstraction, common core system services and a messaging middleware layer. This messaging middleware provides one portion of the plug-and-play attributes of the system, specifically the routing of commands and telemetry data between applications. The messaging layer, also commonly

referred to as the Software Bus, provides a dynamic Publish and Subscribe interface for transferring data between cFE applications. The cFE Software Bus builds upon heritage software bus software developed in the early 1990s. Specifically, it adds the Publish and Subscribe interface which allows message routes between applications to be established at run time instead of at compile time as was previously the case.<sup>2</sup>

In a system developed with cFE, software applications are designed to be loosely coupled. Figure 2 provides simplified diagrams that demonstrate the differences between a loosely coupled plug-and-play system consisting of multiple applications and a single application system consisting of more tightly coupled software packages. As described earlier, the STEREO legacy system consists of a single application made up of many software packages. Interfaces are defined for these packages through public routines contained within the packages. Within this architecture, any software package can call any public routine from another package. All of the links between packages are established at build time.

In contrast, the plug-and-play architecture divides the software system into several smaller applications. Each of these applications performs a unique function and does not provide public interfaces to other software applications. Applications interact entirely through messaging across a software bus. Within each smaller application, the package to package coupling still exists, but on a much smaller scale. Additionally, each application is compiled and loaded separately and



**Figure 2 – Comparison of Legacy System and Layered Plug-N-Play System**

therefore has no knowledge of any other application within the system. Therefore all coupling is through data, allowing greater flexibility of run time change.

One significant difference between the two systems is that commands are handled very differently. Within the legacy system, all command execution can be handled through a single package which in turn calls command execution routines in other packages through public interfaces. Within the publish and subscribe messaging architecture, command execution is distributed to the smaller applications. Each application subscribes to its commands, and commands are then routed to the application through the software bus. It is the responsibility of each individual application to validate and execute the commands which it has received. If new applications are later plugged into the system, they simply subscribe to their commands at run time and they will be routed to the application. The dynamic and distributed nature of this system raises a few obvious questions. First, if commands are sent to multiple applications at nearly the same time, how can the execution order of these commands be guaranteed if required? Second, how can commands from different sources be managed and prioritized such that a single command source can't overwhelm any application?

### RBSP PROTOTYPE COMMAND MANAGER

In order to address these specific questions, the RBSP team created a prototype Command Manager (CM) application which plugs into the cFE and acts as a

mediator for commands to all other applications. The CM draws on the existing data structures from the STEREO command subsystem. Figure 3 shows a high level data flow which details the messages sent to and from various prototype applications. In this diagram, prototype cFE applications are shown as rounded rectangles and command and telemetry messages to and from the software bus are shown as arrows in or out of the cFE as appropriate.

Each of the three command sources identified in the STEREO heritage software has been packaged as a separate application. The Real-Time Command Ingest (CI) application is responsible for receiving real-time uplink commands and forwarding them on for execution. The Time Tag Commanding (TT) application is responsible for storing commands for execution at specified times. The Autonomy Engine (AUT) application generates commands as a result of onboard fault protection rules. Each of these applications has the ability to send macro execution commands.

Now, instead of inserting their commands directly into the command data structures as was implemented in the previous STEREO architecture, these applications simply send the commands to the software bus in the form of a "Command Manager Command". This command contains the end application command, plus additional priority information used by the command manager to ensure proper command execution. The CM application, having previously subscribed to these

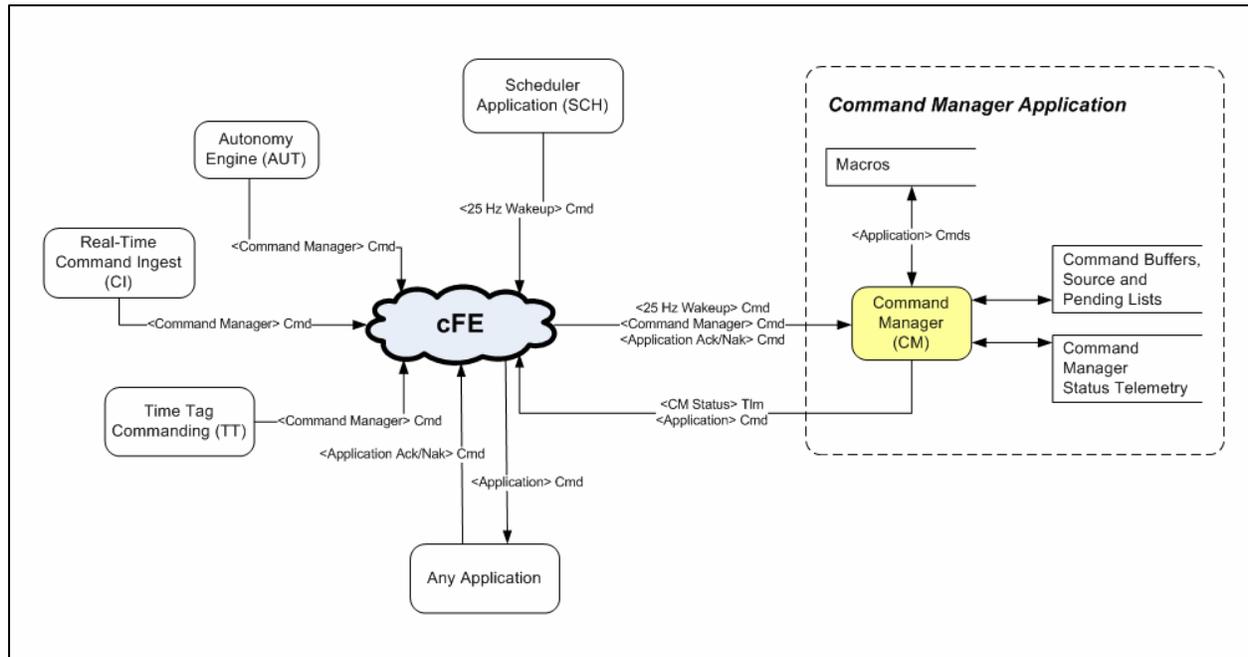


Figure 3 – High Level Data Flow for Prototype Command Manager

commands receives them and updates the command data structures to insert the commands into the command source and pending lists.

Once the command has been linked into the command data structures, the CM application is then responsible for ensuring execution order for the commands. This is accomplished through the use of a scheduled wakeup message. In keeping with the STEREO heritage software, which handled 25 commands per second, the prototype implemented a 25Hz wakeup. At initialization, the CM application sends a schedule request to the scheduler (SCH) application and then waits for receipt of the scheduled wakeup. Upon receipt of this wakeup command, the CM application will send the next pending "Application command" onto the software bus. After the command is executed, the target application will respond with a command acknowledge. This acknowledge will indicate if the command was successful or in error. Upon receipt of the acknowledge command, the CM application will update its command data structures and status

telemetry. If an acknowledge command is never received by the CM application, the command will "time-out" based on a programmable time out period. To implement this, the CM application uses a command sequence counter to ensure that the command acknowledge corresponds to the most recent command. This command and acknowledge handshake is most easily shown in the sequence diagram in Figure 4.

The CM application also implements the commands to execute macros and manage command priorities. When a macro is started, the CM application maintains information about the source of the macro execution command and the priority at which it is executing. This allows for macros to be aborted and suspended by command priority level as well as to be aborted by command source, thus providing the same capability as the heritage STEREO system.

Lastly, the CM application also maintains command history and provides command execution status telemetry. The command history contains the CCSDS

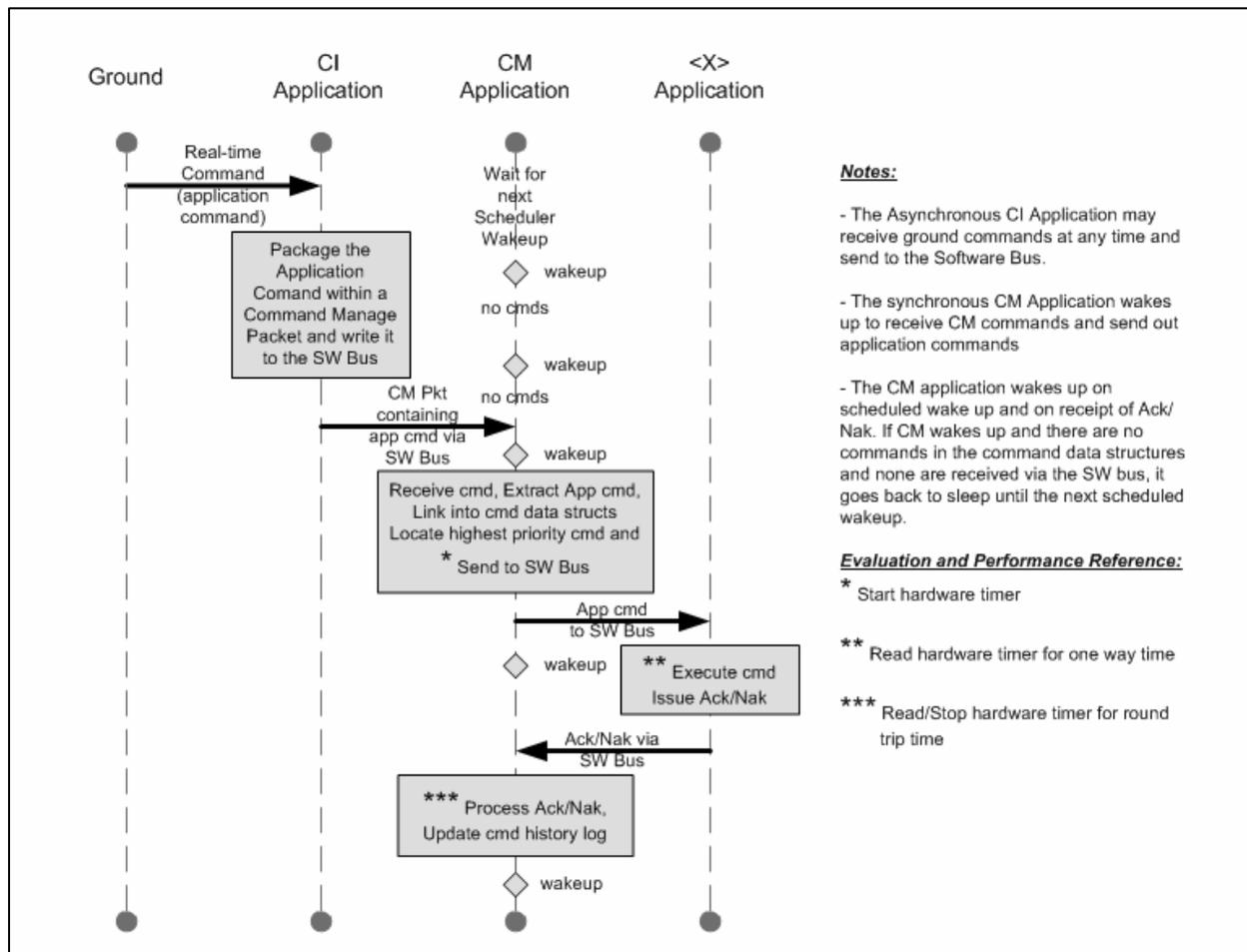


Figure 4 – Command Manager Message Sequence Diagram

command packet header information of each application command and shows when the command was received and routed on the software bus to its target application. It also contains the CCSDS packet header information of the “command acknowledge” packet and shows when the command acknowledge was received. In addition to the command history telemetry, the CM application provides housekeeping telemetry that shows the numbers of Real-time, Time-tagged and autonomous commands that have executed, and whether these commands were successful or in error. This housekeeping telemetry indicates whether there are any pending command acknowledges. It also provides detailed information on the last command executed and information on the state of command macros.

Some functionality of the heritage STEREO system was not implemented in the CM application prototype. For example, the command checking functionality that is normally performed when commands are uploaded to the system is not performed. Instead, it is the responsibility of each target application to perform the necessary command verification at execution time. The ability to perform additional command verification at uplink time could be incorporated, but it would require each application to implement command verification independent of command execution.

## EVALUATION AND PERFORMANCE

Upon completion of the CM prototype application, a qualitative assessment of the system was performed. The prototype CM application provides the same command management functionality as the existing software. It also has the additional benefit of being decoupled from all of the other flight software applications. This allows the CM itself to be plugged in and out of the system without significant impact to the other applications. This flexibility allows development of the other software applications without the CM application being present in the system. In this environment, the CI, TT, and AUT applications simply send the target applications’ commands directly to the software bus, bypassing the CM application. Later, if macro capabilities or command prioritization is required, the CM application can be plugged in and the functionality can be added.

Additionally a series of measurement tests were performed on a BAE Systems, RAD750 © processor board. This board was configured to set the processor clock to 33 Mhz, with the cPCI clock set to 16.5 Mhz, which emulates the processing requirements being considered for the RBSP program. These measurements were intended to determine critical timing for command execution and to capture how

much overhead is associated with the command messaging, versus the heritage direct routine call approach.

The following measurements were made: (1) One way time: the bus transaction time to send a command from the CM application to a destination application. (2) Round trip time: the time required for a command to go from the CM application across the software bus through command execution all the way to the receipt of the acknowledge command by the CM application. (3) Ack-time: the difference between measurement 1 and measurement 2 yielding the approximate time between the destination application command receipt and the CM application acknowledge command receipt. The starting and sampling of the hardware timer for these measurements is illustrated in Figure 4.

These measurements were taken under various application task configurations, the first of which included sending no-op commands, (which have no data payload in the command packet) to a synchronous application task whose task rate is dictated by the SCH application. Only round trip command time was measured in this test which yielded several samples closely packed around a small number of discrete points ranging from 400  $\mu$ s to tens of ms. These results are explained by the difference in the application tasks scheduled rate which dominates the measurement.

The second round of timing measurements were taken for the same task after having been modified to pend, not only on its scheduled wakeup but also the receipt of the application commands to which it has subscribed. Again, only the round trip command time was measured, which includes the command and command acknowledge bus transactions, a certain amount of task execution time for each of the two tasks as well as OS task context switching. The round trip time for commands ranged between 400  $\mu$ s and 900  $\mu$ s. These results were not controlled for higher priority application task interruptions or other task activities such as periodic generation of telemetry which likely explains the variability.

The third round of timing measurements were taken using no-op commands for the CM application itself. These commands follow the same path as application commands destined for other applications; however, the minimum measurements likely occurred without task context switching. Within this series of tests the round trip time of the command was measured as well as the one way and the ack-time. The one way time was a relatively consistent 100  $\mu$ s. The round trip time was a relatively consistent 500  $\mu$ s. The ack-time component, being 400  $\mu$ s is likely dominated by routine task

processing within the CM and not due to the cFE. Within this round of measurements, 4 different sizes of CM no-op commands were used (1) the traditional no op consisting of just the CCSDS packet header with no payload, (2) a 64 byte no-op, (3) a 256-byte no-op and a (4) 1000 byte no-op (sized to avoid segmented transfer frames in the prototype in which the CCSDS segment layer was not implemented). The size of the no-op did not significantly effect the timing measurements.

## CONCLUSIONS

The evaluation of the prototype design and performance shows that this architecture provides a much more flexible development environment and less coupling. This is demonstrated by the ability to develop applications independent of each other and then plugging the applications together, with only data dependencies and without compile time coupling. However, the target applications must perform additional processing if additional command verification requirements are imposed. If commands need to be validated at load time, the target applications would need to add separate command verification and additional protocol would need to be implemented so the applications could distinguish a request to validate the commands from a request to execute them.

Additionally, the measurements performed on this prototype application show that the command throughput for this system can easily support command rates equal to the heritage STEREO software without significant processing overhead. It also shows that the amount of command data being transferred across the software bus does not significantly impact processor utilization. This was demonstrated through testing with both short and long command messages. With the typical command uplink rates and typical autonomous or time-tagged commanding rates it is estimated that the command management will utilize a very small percentage of the CPU.

Finally, these measurements show that in order to implement a closed loop command system, with command acknowledgement, the design of the target applications can have a significant effect on overall system command throughput. One conclusion that can be drawn from this is that the most efficient system for command execution with the centralized Command Manager should contain target applications that are more data driven, pending on commands. If target applications are independently scheduled, the execution time of commands is driven by the scheduling of the applications and undesired delays can be introduced into the system.

## ACKNOWLEDGEMENTS

The authors would like to acknowledge the STEREO and RBSP projects for supporting this work. We would also like to extend appreciation to everyone within the Embedded Applications Group who contributed to the prototyping efforts, either through direct prototype development or through supporting reviews of our work and providing feedback. We would also like to thank the cFE team at NASA GSFC for their openness and willing support.

## REFERENCES

- [1] Mission Guide, "The Sun in 3-D: A New Frontier in Solar Research", Johns Hopkins University, Applied Physics Laboratory, October 2006.
- [2] "Implications of Responsive Space on the Flight Software Architecture", 4<sup>th</sup> Responsive Space Conference, Jonathan Wilmot, NASA Goddard Space Flight Center, April 2006.

## BIOGRAPHIES



**Christopher A. Monaco** is an Associate Staff member 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 Command and Data Handling (C&DH) flight software as the Lead Flight Software Engineer of the C&DH and Earth Acquisition (EA) software applications on the STEREO mission. Chris has 6 years of experiences in real-time embedded flight software systems on two NASA missions. Chris has a B.S. in Physics from Clarkson University, a B.S. in Computer Science from Clarkson University and a Master's of Science from 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 17 years experience in development of flight software for various NASA missions. His experience includes Guidance and Control, Command and Data Handling, Power Subsystem and Autonomy Subsystem software. Mark is currently the flight software lead engineer on the Radiation Belt Storm Probes (RBSP) mission. He received his undergraduate degree in mathematics from Western Kentucky University and has an M.S. in Computer Science from the Johns Hopkins University.