

**ACTIVITY NODE BASED FLIGHT SOFTWARE AS A
BENEFIT TO SYSTEMS ENGINEERING**

**A Thesis Project
Presented to the Faculty of
California Polytechnic State University
San Luis Obispo, CA 93407**

**In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Aerospace Engineering,
Specialization in Space Systems Engineering**

**By
Eugene Daniel Lewis**

March 2012

© 2012

Eugene Daniel Lewis

ALL RIGHTS RESERVED

Committee Membership

TITLE: ACTIVITY NODE BASED FLIGHT SOFTWARE AS A
BENEFIT TO SYSTEMS ENGINEERING

AUTHOR: Eugene D. Lewis

DATE SUBMITTED: March 2012

COMMITTEE MEMBER: Dr. Eric Mehiel
Cal Poly Advisor, AERO Department

COMMITTEE MEMBER: Kira Abercromby
Faculty, AERO Department

COMMITTEE MEMBER: Dan Wait
Lecturer, AERO Department

COMMITTEE MEMBER: Dr. Jerry F. Drake
Industry Advisor

Abstract

ACTIVITY NODE BASED FLIGHT SOFTWARE AS A BENEFIT TO SYSTEMS ENGINEERING

By Eugene Daniel Lewis

This report discusses one application of a flight software design for a spacecraft in which the software executes from a database that can be managed by systems engineering.

This report gives an overview of how such a software design can be developed and implemented. It also discusses why this approach is beneficial to the systems engineering program.

Preface

The opinions expressed in this document are my own. The scope of this report is to outline the design of a database driven flight software architecture and discuss the benefits to systems engineering.

Acknowledgements

I would like to acknowledge and show my appreciation for the support, either directly or indirectly, that I have received for this thesis project and report. I would first like to thank Phil Shirts, Flight Software CPE for the NIRCcam Program, for the flight software design concepts this report was based on. I would also like to acknowledge Tanya Kruglikov for helping with the concept of the ground based systems proposed. I am grateful for the assistance of Jerry Drake in being my advisor at LMSSC, and to Professor Eric Mehiel as my Cal Poly advisor. A very special thanks to my wife and family for their understanding and support while I was pursuing this degree.

Table of Contents

Table of Figures	viii
1. Introduction	1
2. Background	4
3. Activity Nodes	9
3.1. Activity Node Definitions	14
3.2. Activity Node Structure.....	15
3.3. Runtime Modification of Activity Nodes.....	17
3.4. Software Command Engine Overview	21
3.5. Command Staging	24
3.6. Command Execution	26
4. Systems Engineering	28
4.1. Ground based tools.....	29
4.2. Activity Node Editor	30
5. Conclusions	32
5.1. Advantages/Disadvantages.....	32
5.2. Future Directions	34
5.3. Conclusion.....	36
6. Bibliography	38

Table of Figures

Figure 1: Example Turn On Gyro Flow Diagram.....	10
Figure 2: Example Turn On Gyro Sequence Diagram.....	11
Figure 3: Example Turn On Gyro Activity Node Sequence.....	12
Figure 4: Updated Turn On Gyro Activity Node Sequence for 90s Delay Time	13
Figure 5: Updated Turn On Gyro Activity Node Sequence for 'Test' Mode.....	14
Figure 6: Example of an Activity Node Data Structure	15
Figure 7: Different Approaches to Defining Activity Node Structure	17
Figure 8: Example of Activity Node Sequence Runtime Modification.....	18
Figure 9: Example of Activity Node Method Modification	21
Figure 10: Software Command Engine State Diagram.....	23
Figure 11: Activity Nodes copied into Staging Area.....	26

1. Introduction

Over time in the spacecraft industry, flight software has supplanted hardware as the control element of spacecraft flight management and is playing an increasingly important role in spacecraft systems. As flight software has grown in this role, it has become more and more complex. This increase in complexity has made it difficult for programs to manage, develop, and execute software designs for flight systems. In a report commissioned by NASA it was found that engineers and scientists don't realize how their decisions in the design affect the downstream complexity of the flight software system³. In the industry it has been stated that a good development process can reduce the complexity of the software and minimize the number of defects in the flight software.

Flight software has also had an increase in the involvement in spacecraft accidents. An Aerospace Corporation study of failures that occurred between 1998 and 2000 estimated that half were related to software². Many of these failures were due to such factors as underestimating software risk, poor communication and information flow, unnecessary complexity in the software and system, inadequate review processes, not enough software testing at the system level, and other areas attributed to systems engineering. For any system as complex as a spacecraft good systems engineering is essential for success and must have a more defined role in the flight software process.

Historically in spacecraft systems there has been a disconnect between the development and execution of flight software from the other parts of a satellite

program. Often software engineering is treated as a separate group with little interaction with the rest of the subsystems. A major task of systems engineering in spacecraft design is to manage the interfaces between all the subsystems including software. Systems engineering develops a design and creates requirements but they are often not well defined for flight software. This is often due to the growing complexity of both the spacecraft system design and the flight software. The result is flight software is often delivered behind schedule and not up to the expectations of the program. There is a need for systems engineering to become more involved in the flight software design and execution. A major issue with this is most systems engineers are not trained to be software engineers and have little understanding of how software behaves in a real-time system. On the other hand most software engineers do not have the background of space based satellites or hardware engineering.

This paper describes one solution to bridging the gap between flight software and systems engineering. The area of flight software development addressed is command execution algorithms. The operational commands of a satellite are designed and developed by systems engineers. The algorithms are then coded into flight software by software engineers and often get implemented differently than the systems engineers intended. This is often the case because of the inadequate involvement of systems engineering in the software engineering process. The solution proposed is a way for software engineers to develop a software architecture and leave the details of the command algorithms to systems engineering. This allows systems engineering to

develop and create the software command algorithms themselves since they are the most knowledgeable about how the commands are to be implemented.

2. Background

Flight software has a history of developmental problems in the aerospace industry. There is also a perceived rift between systems engineering and flight software development. Too often the flight software is developed and delivered without much involvement of systems engineering. The software development plan is created by software engineers and approved and put into motion. It isn't until later that systems engineering gets involved and starts to find inadequate design flaws that cause the software to be constantly redefined. Sometimes it isn't until after the fact that systems engineering gets involved and finds the delivered software is inadequate to support the requirements of the program. These differences are a main contribution to flight software being historically late to deliver and over budget.

One major task of the flight software subsystem is command processing. These commands are sent from the ground or uploaded into tasking request packets that trigger the flight software to perform specific algorithms. The number of commands varies depending on the satellite system but often can be in the several hundred command range. Each command must have an algorithm defined in the flight software to perform when the command is received. Although these algorithms are often defined by systems engineering they are implemented by the flight software developers.

One of the primary issues with flight software and command algorithm processing is that during the development of the system (and often operations) modifications to the

algorithms are required. In normal flight software systems these modifications require partial or even full flight software updates. Every flight software update takes time since the software needs to be developed, tested, certified, and released before it can be installed on the spacecraft.

To complicate matters, the software engineers who develop the flight software are trying to implement algorithms defined by systems engineers who often do not understand software. Conversely, the algorithms are developed and implemented by software engineers who may not fully understand the day to day operations of a satellite. The type of software that is developed for home computers or internet website applications is entirely different than the type of software needed to control a satellite. On the other hand systems engineers are not fully knowledgeable about software development. Even when they do work together, software and systems engineers are coming from two different backgrounds and often fail to converge on the best solution to fit the needs of the system.

The industry is just starting to realize that system engineering needs to be more involved with flight software development. Today the term “Software Systems Engineering” is becoming more used in the aerospace industry. This is a dedicated group of systems engineers that have a background in flight software. Software systems engineers are a start in the right direction to find a way to develop flight software that meets the needs of the entire satellite system.

Systems engineering involvement with the development of the flight software from the start of the architecture design to the final testing and delivery is the best way to improve the flight software process. The challenge has been getting the systems engineers and keeping them engaged during the software development. One solution is to develop software in a way that systems engineering can be more involved.

Flight software in space systems is a different type of software than is normally developed for applications. Flight software is a real-time operating system and must respond to signals and triggers. These systems are often referred to as event driven systems. Flight software also must be a multitasking system since it must handle and control several different tasks simultaneously. On many spacecraft systems a single flight processor is used and therefore only one task can have control of the CPU at a time. When a task has control of the CPU it cannot 'block' or the system may miss an important or critical signal requiring a service. Each task must complete and relinquish control to another task in a very small amount of time.

This thesis project is a design of flight software where the command processing is reduced to an engine that performs generic tasks given a set of parameters. The parameters are arranged into database elements that can be controlled and managed outside the flight software. The database elements are called activity nodes. The activity nodes store the information needed for each task and are executed sequentially as defined in the database. The tasks performed by the activity nodes are simple basic tasks in the flight software that require a short time to process. While the flight

software developers would be responsible for the command processing engine, the ownership of the activity nodes would be systems engineering. This gives systems engineering direct control of the algorithms performed for the commands. The flight software command processing function now behaves like a database driven software routine.

Database driven or table driven software is not a new concept. It is found in all types of applications, including many websites. This type of software is sometimes used in object oriented programming. Flight software real-time systems are also object oriented software architectures. In this type of design, the program code is constant and stays the same during a routine, and the data set it processes controls the flow. There is a main software executable, an inference engine, that processes database elements sequentially to perform an overall task. For this application of using an activity node database to process flight software commands, the activity nodes are the database elements that control the flow of the command sequences.

The activity node flight software design requires an inference engine to process the activity nodes. This is called the command sequence interpreter. An inference engine is software that executes instructions from a knowledge base. The main elements that make up an inference engine are an interpreter, a scheduling routine, and a consistency checker. In the case of the command sequence interpreter, the interpreter processes the information stored in the activity nodes into software instructions. Activity node timers handle the scheduling of when the interpreter processes the activity. The

command sequence interpreter also performs a consistency check before trying to execute the instructions of the activity node. Inference engines are frequently the drivers of data-driven systems such as this activity node flight software design.

The idea proposed in the paper is to show how a table driven database implementation of flight software can be beneficial to a spacecraft system. To explain this concept it is important to discuss a simple example of such a system. The concepts discussed in Section 3 are a broad overview of how such a system can be used in a full spacecraft system. Simple examples are used, specifically turning on a gyro and moving a mechanism. These are very simple generic examples and were used to keep the concepts here simple. This concept is not new or invented, just adapted from existing software applications to examples used in aerospace designs. The implementation of this activity node system would have to be greatly modified to be used in a real spacecraft system.

3. Activity Nodes

Every command received by the flight software executes a specified algorithm to perform its function. These algorithms are broken down into sequences of small primitive operations. Each primitive operation can be classified into a specific type of software instruction. For example, a sequence to turn on a gyro would be comprised of interface commands, telemetry requests, telemetry verifications, and perhaps some wait time delays. These algorithms are often designed by systems engineering and implemented in the flight software. Take an example of powering on a gyro upon receipt of a command. The flow of this process could be diagrammed into the flowchart shown in Figure 1.

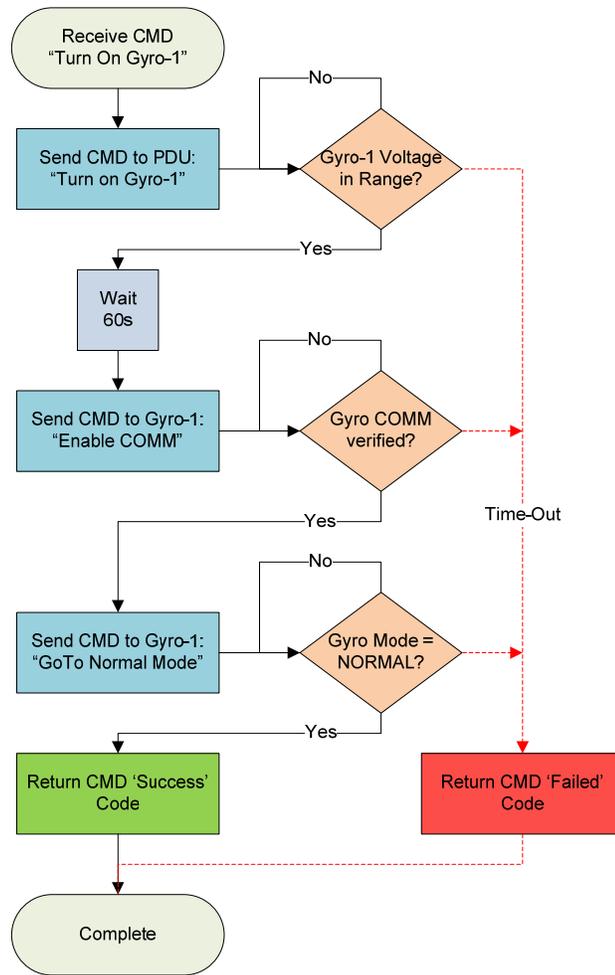


Figure 1: Example Turn On Gyro Flow Diagram

For flight software implementation of the above process, the events are broken down into single instructions inside the software. Some of these events require communication on data interfaces to other components on the spacecraft. Other operations are internal to the flight software. Event driven sequences can usually be diagramed in a software sequence diagram. Figure 2 below shows a sequence diagram of the example sequence to turn on a gyro.

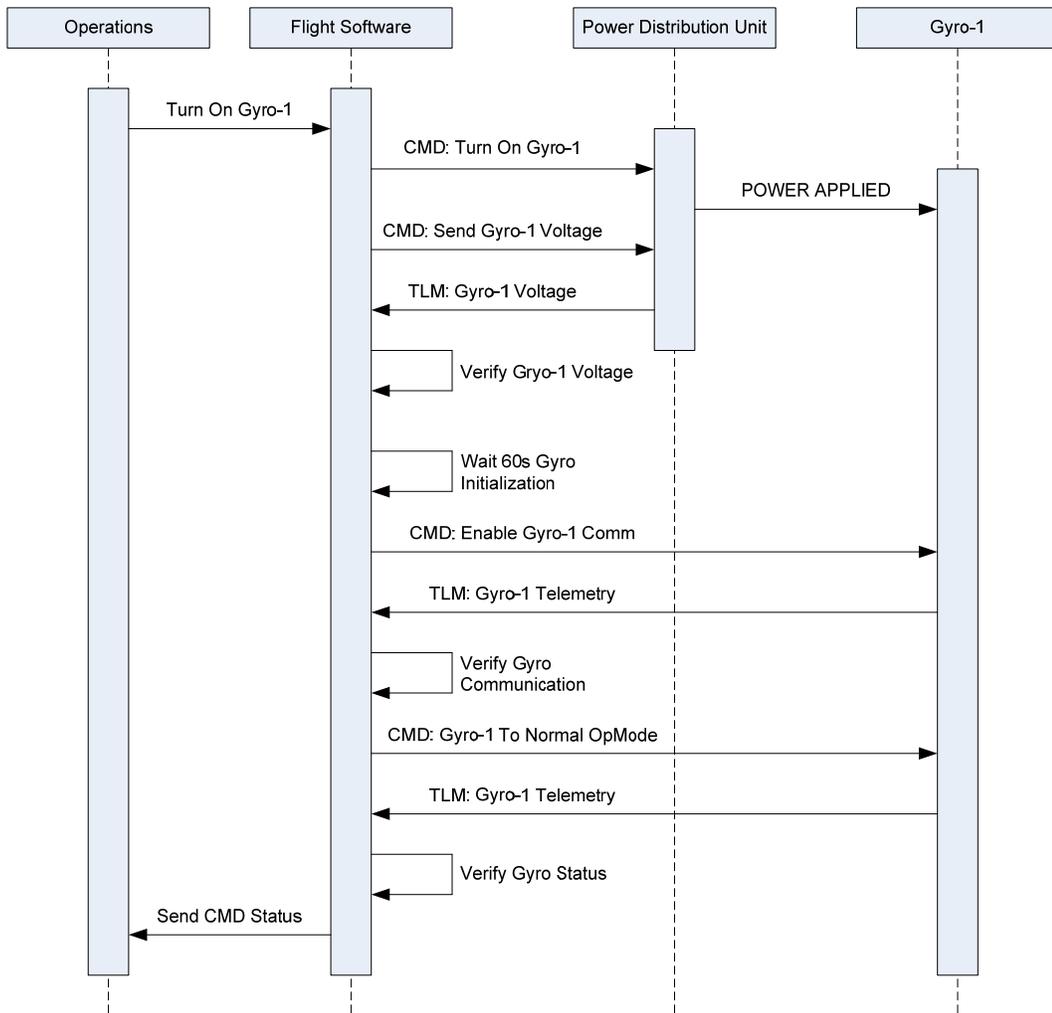


Figure 2: Example Turn On Gyro Sequence Diagram

Activity nodes are data structures that contain information about a software instruction to be executed in a runtime environment. Each node is configured for which type of instruction is to be performed. If it is an interface command, the command parameters are configured inside the node. If it is telemetry verification, the details on which telemetry point and range to verify against are configured in the activity node. The

activity nodes are then executed sequentially to perform the command algorithm.

Figure 3 below shows how the turn on gyro sequence can be defined by activity nodes.

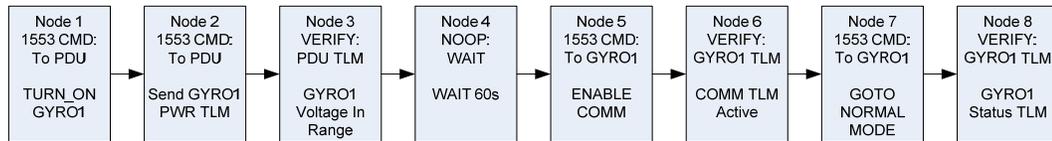


Figure 3: Example Turn On Gyro Activity Node Sequence

Both the sequence diagram and the activity node diagram break down the sequence to turn on a gyro into a series of sequential primitive events managed by the flight software. The activity nodes give the user the ability to model the command sequences into individual database instructions. Since the activity nodes can be classified into types the flight software only needs to provide a method to execute these nodes. The flight software needs to design an interpreter to ingest each activity node configuration. This engine must then be able to execute the instructions contained in the activity node. For each type of activity node, the flight software will perform that instruction given the parameters defined in the activity node.

The ability to break down commands into basic instructions and store these in onboard table gives the outside user the ability to manage and update command algorithms outside of the flight software. All commands can be mapped to their own unique activity node sequence. These activity node sequences can be stored together in a table in the flight software. If changes need to be made to the algorithm, activity nodes can

be modified, added, or deleted in the onboard table. By placing command algorithm changes into tables the need for flight software updates is significantly reduced.

From the previous turn on gyro sequence example, over time a gyro may require more time to initialize. A change is required to the algorithm to wait 90 seconds instead of 60 seconds before enabling communications. Normally this change could require a flight software update. Flight software updates on orbit can be a complicated operation. However a new table load changing the wait-time parameter in node 4 of the activity node sequence is a simple poke to an onboard table. The resulting sequence is now shown in Figure 4.

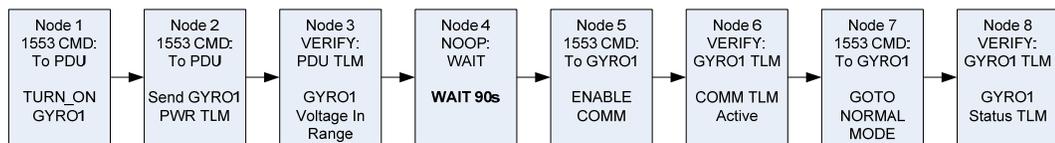


Figure 4: Updated Turn On Gyro Activity Node Sequence for 90s Delay Time

Another example using the turn on gyro sequence is during integration and test at the factory, the gyro cannot be put into ‘normal’ mode without causing faults to the system. To avoid this, the gyro manufacturer built in a ‘test’ mode command to use instead. Now during the integration and test phase a version of the table can be made changing node 7 to command ‘test’ mode and node 8 to verify ‘test’ mode status in the gyro telemetry. The test version of the turn on gyro sequence is shown in Figure 5. These

kinds of updates allow integration and test operations to use the same version of flight software that will be used for mission operations on-orbit.

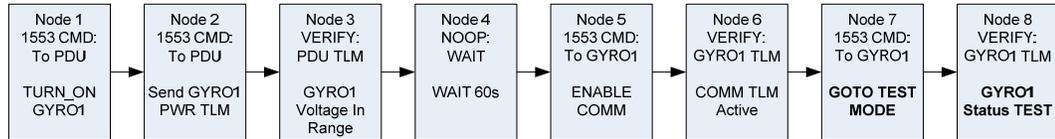


Figure 5: Updated Turn On Gyro Activity Node Sequence for 'Test' Mode

3.1. Activity Node Definitions

Different activity node types are needed for different flight software instructions.

There should be a node type for sending a command to a component. Another node type should be created for verification of a telemetry value for a specified range.

Another type can be defined to access memory locations for table values or telemetry packets. Other types may include a request for data from a component, enabling or disabling a protective measure fault, or running an internal software routine.

The definition of the activity nodes can be kept at a higher level (i.e. command type) or broken down into more specific lower types (i.e. 1553 command type, spacewire command type, discrete command type). It is up to the system architecture to decide what activity node types need to be created.

Even though there are several different tasks it is highly recommended to map the activity node types to a single software instruction. This keeps the building blocks

approach to making software algorithms very simple. There may be times when combining types together may be required. For example the software may be required to keep sending the same command until a specific response is seen. This could be done with combining a command node with a telemetry verification node. It would send the command and check the telemetry, and if the telemetry failed it would repeat the command and telemetry verification again.

3.2. Activity Node Structure

Each activity node needs to contain a specified set of information. This information is required by the flight software interpreter to know how to use the information in the activity node. These activity node fields include which command this activity node belongs to, where it is in the sequence, and what type of activity node it is. These fields may also include a description string, how long to wait before continuing on to the next activity node, and any other information fields required for activity node execution. A sample of this data structure is shown in Figure 6.

Activity Node Data Structure		
<u>Parameter</u>	<u>DataType</u>	<u>Example</u>
Cmd Number	int	44
Node Number	int	1
Node Type	uint8	CMD
Description	char[80]	"Send Turn on Gyro cmd to PDU"
TimeToNextNode	int	5
Command Type	uint8	1553
Command Data	uint16[5]	55d3 4001 0000 0000 0000

Figure 6: Example of an Activity Node Data Structure

In addition to general activity node information, each type of activity node requires a unique set of data fields for its interpretation. A command node may contain a destination code and command data fields. A telemetry verification field may contain what data packet and offset of the data to check and the passing range of the value. Depending on the design of the flight software interpreter and the allowable size of the table these parameters can be required in all nodes or just of nodes of that activity node type.

It may be advantageous to keep all fields the same for all activity nodes. This would mean all activity nodes follow the same definition. Ground operations would not have to track different definitions of the activity node fields because all activity nodes would be of the same structure. The disadvantage of this approach is the activity nodes are sized larger than they need to be because a command node would have unused telemetry verification fields.

An alternative design is each activity node type has its own unique data structure definition. This could result in different sizes of activity nodes. If onboard memory space is really an issue this would be the best approach.

Perhaps the best approach is a way to combine these two ideas. This design would use the maximum size required for an activity node definition and then size all activity nodes to that size, using 'spare' fields to fill out the size. The activity nodes would all be the same size and allow better management of the flight software tables and ground management tools. It also allows ground tools to better parse table dumps of the

activity node database since all nodes are still of the same size. Keeping the size constant for all activity node types would be the recommended approach for usability. Figure 7 shows examples of different ways to define the activity node structure.

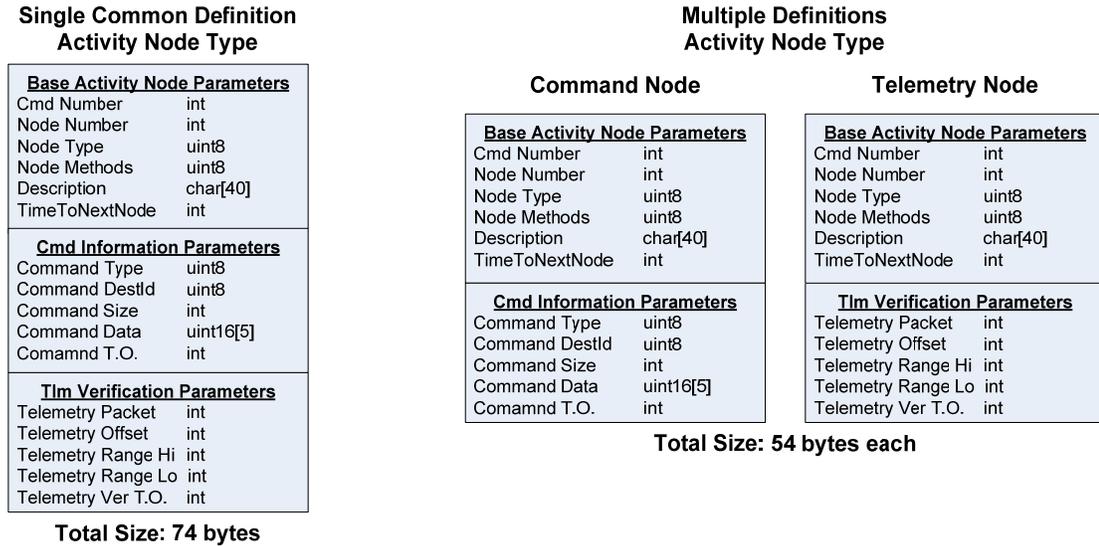


Figure 7: Different Approaches to Defining Activity Node Structure

3.3. Runtime Modification of Activity Nodes

It is not always possible to predefine static values of the command and telemetry parameters stored in the activity nodes. Ground commands are often formatted with variable parameters. As an example, a command to control a deployment mechanism has a variable parameter that specifies 'deploy' or 'retract' cases. The flight software would receive the command data and have to modify the algorithm sequence per this variable parameter. The two sequences for 'deploy' and 'retract' are very similar with

only a few minor differences. The 'deploy' case requires enabling mechanism power at sequence start and formatting the command data to the mechanism motor for a 'deploy' case. The 'retract' case would instead format the command as a 'retract' case but skips the mechanism power at the start because it is already on from the 'deploy' case. It would instead need to turn off mechanism power at the end of the sequence. The same activity node sequence can be used for both cases as long as there is a way to modify the activity node before executing the instruction. Figure 8 shows an example of the original activity node sequence as compared to the two different runtime sequences.

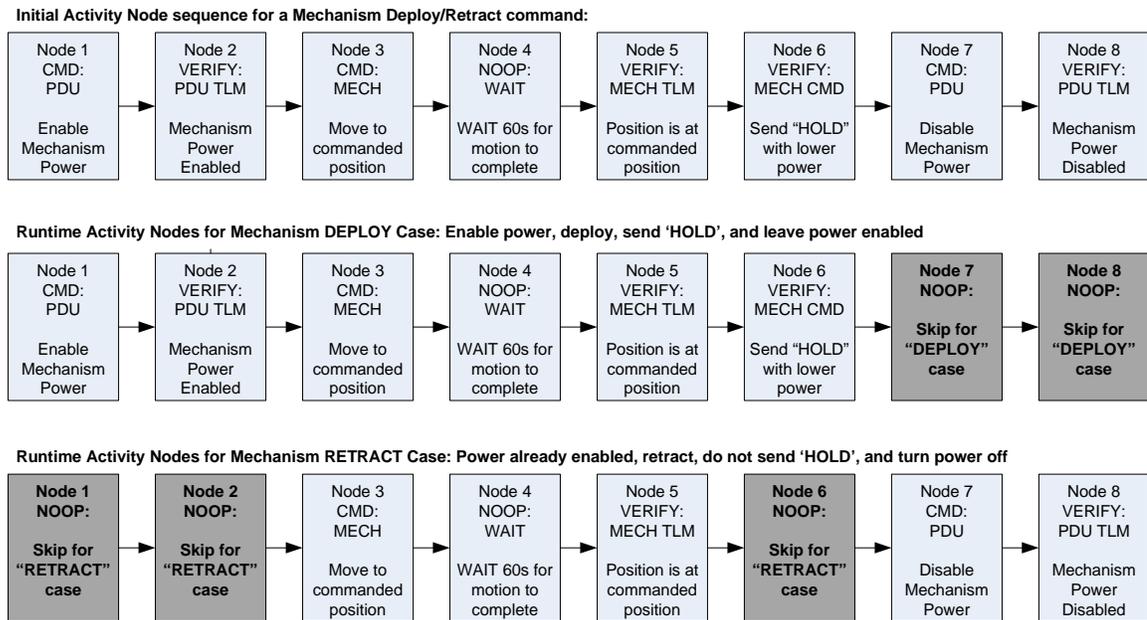


Figure 8: Example of Activity Node Sequence Runtime Modification

The activity modes can be modified by using a set of defined methods that perform real-time functions in the flight software. The way to run one of these methods is to configure a field in the activity node that signals which method to run. Since there are several different commands there needs to be a full library of methods that can modify the activity nodes. The library can be narrowed down for every command into a usable subset required for that specific command. For example, the command to deploy a mechanism does not require access to methods that configure gyro data. The available methods from the method library can be set on a command by command basis.

To access the available methods, a 'map' that assigns each method a unique value to call must be defined. The method that formats the command data for a 'deploy' or 'retract' case could be assigned to method value '1'. The method that determines whether to run or skip the enable mechanism power command can be method value '2'. Now a parameter can be used in the activity node that can execute a specific method before the node is executed. This requires a type of method router that maps each method value to a call of a method from the method library.

Sometimes there may be more than one method required to configure an activity node. In this case, a 'mask' of methods would be the recommended solution. So if the activity node needs to run methods 1, 4, and 7, it can configure a bit mask with those three bits sets that would execute all three methods before node execution. Sometimes one method may need to be run sequentially before another method. This is another advantage of using a 'mask' of methods where the mask can be configured to

sequentially run methods in a specified order. So if method 1 always needs to be run before method 2, the method routing routine would check bit 1 first, then bit 2, then the remainder of the mask.

Using the turn on gyro sequence example, the ground command to turn on the gyro has a variable parameter selecting 'GYRO1' or 'GYRO2'. Figure 9 shows the initial activity node sequence stored in the activity node table and how the method router is configured with four methods to configure the activity nodes. The methods required to be run are shown in the initial sequence where activity node 1 requires method 1, activity node 3 requires method 2, etc. The two different resulting activity node sequences are shown below when the command parameter is 'GYRO1' vs. 'GYRO2'.

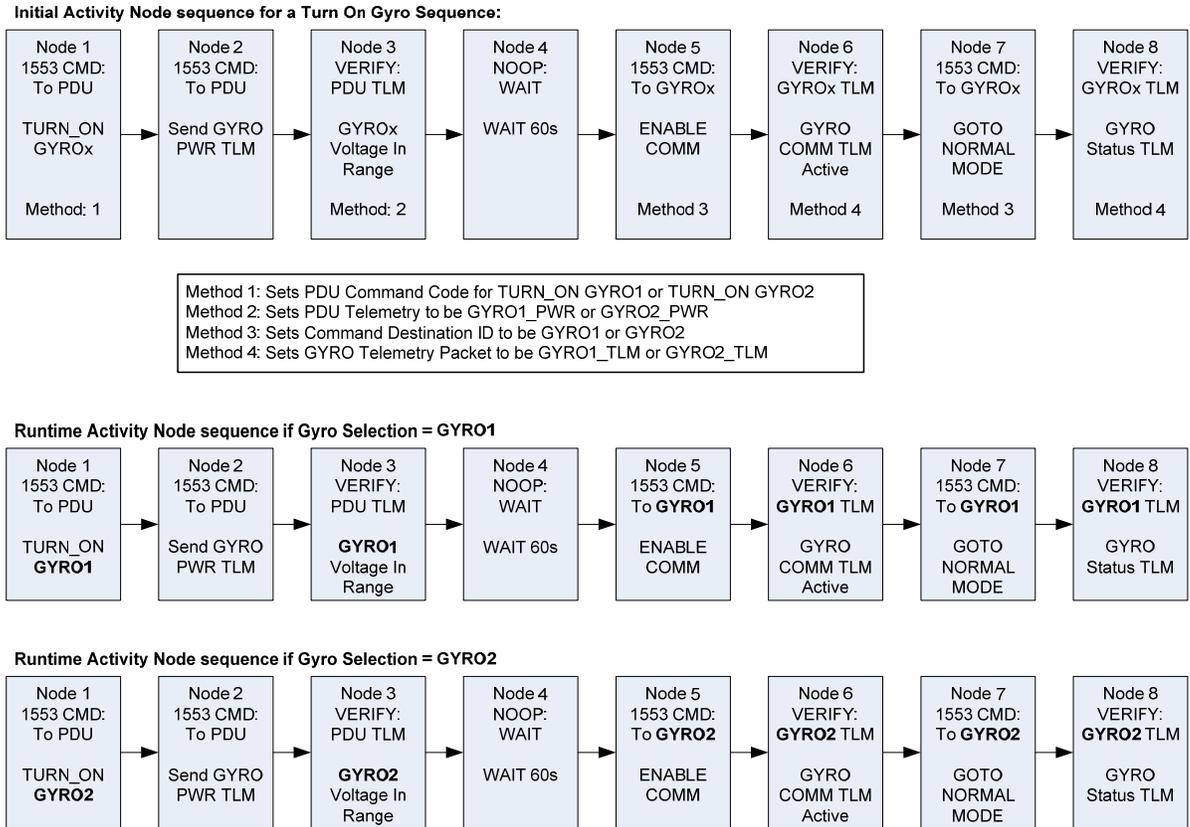


Figure 9: Example of Activity Node Method Modification

3.4. Software Command Engine Overview

Activity nodes allow flight command algorithms to live in a database instead of coded into flight software routines. This reduces the command processing in the flight software down to a processing engine of activity nodes. Processing of the activity nodes is just a routine of performing the task of the node type with the information in the node, then moving to the next activity node until the activity nodes are completed for

the command. When executing an activity node, the software will execute the task matching the activity node type using the parameters specified in the activity node.

Flight software in satellite applications is often built as a runtime event driven system.

One of the advantages of developing the activity node engine is it gives the software an opportunity to return to the normal operating state between each node execution before processing the next activity node. This is most useful when the command processing is running on a single thread of execution. Each node performs a simple short task and returns to the normal state. When commands require a long sequence of operations, this means the software is given many chances to return to the normal state in case there are other signals to process during command execution. Then it can continue onto the next activity node in the sequence. This means on a single thread of execution the flight software will not be put into a 'blocking' state and ignore other events when process a command algorithm.

For our example, we will treat the flight software as accepting commands while in a normal 'engineering' state. When in this state, a command will trigger the flight software into a transition to operate on the received command. Then after each activity node execution, the flight software will set a new signal to itself with a time delay before processing the next node. This time can be specified in the activity node, which is done when pauses in the sequence are needed. When delays are not required, setting a default short delay is recommended to give the software time to be in the normal state between nodes. This delay can be in the millisecond or microsecond range

depending on the system availability or requirements. The flight software command processing can be built as in the UML diagram in Figure 10.

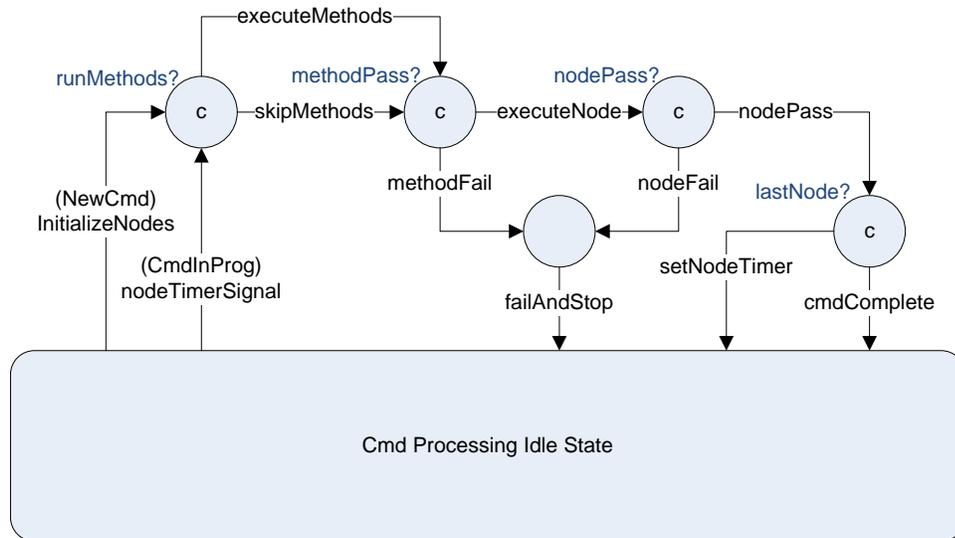


Figure 10: Software Command Engine State Diagram

As shown in the above example, when a new command signal is received, the flight software will initialize the activity nodes for the matching command signal. It will then run any setup methods before executing the node. After the node activity is performed, if there are more nodes to run it will set a timer to start the next node then return to the idle state. When the node timer expires it will then process the next node by running any setup methods and executing that node activity. This will continue until either a node fails (failAndStop) or the last node is performed (cmdComplete).

Since each timer is independent as to when it may fire, multiple commands can be run simultaneously using this type of software command processing. Each timer would

trigger its activity node to run and after completion would return to idle state to wait for next timer. This is very important when the software needs to simulate running parallel command processing in a single thread of execution.

3.5. Command Staging

When the activity nodes are configured to run, the data inside the node can be modified. The next time the command is executed, the activity nodes need to be reset to the initial state. To prevent this, the original activity nodes need to be left untouched. This means the activity nodes need to be copied into a temporary area of memory. Then the nodes can be modified and executed in this temporary memory area. This area of memory is allocated for the task of copying and executing command activity node sequences. This area can be referred to as a 'staging' area. When the command is complete, the area is reinitialized and the next command can be copied into its place. The use of the command staging area is shown in Figure 11.

If the software needs to support parallel command processing, then multiple staging areas need to be allocated to hold multiple command sequence nodes simultaneously. The area does need to be allocated by the size of the largest command to be held in the area. This can either be a single large area that can hold separate activity node sets, or an array of smaller areas. The advantage of using a large area is that it can occupy less memory overall. For example, if the largest command is 50 activity nodes long then the

staging area can be allocated to be a single area able to support 100 nodes. Then if three more commands are started at 10 nodes each, this area of 100 activity nodes can support all 5 commands. But if an array of staging areas is to be used, then each area would need to support 50 nodes. So the same example would require an area of memory capable of holding 250 nodes. The disadvantage of using a single area is it requires management. If a new command is received and there isn't enough room left in the allocated memory, it would need to be either rejected or queued up. It may also be confusing to the user as to why sometime the flight software can handle several commands simultaneously and other times only a couple at a time.

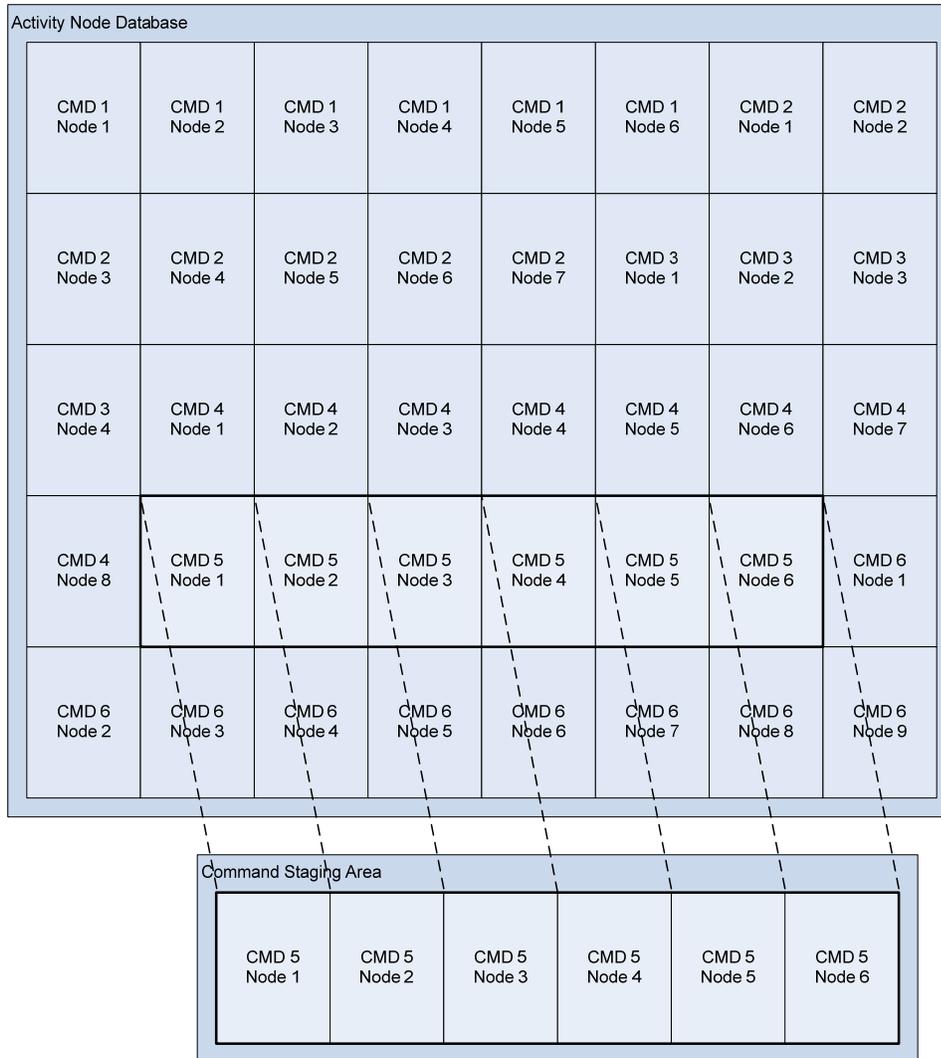


Figure 11: Activity Nodes copied into Staging Area

3.6. Command Execution

Once a command is copied into the staging area, each activity node is then executed.

Upon execution, any methods that are required to run to configure the activity nodes

must be performed before the node is executed. Because these methods are executed

in the staging area, the original activity nodes are not modified and preserved for the next execution of the command. If any of these methods fail to configure the node, the command processor engine must stop and return a failure before the command can start. This would prevent a command from starting up and failing in the middle of a sequence.

Setup Methods are executed on the activity node in the staging area and don't modify the original memory area. Then the node is executed. If successful, it will then continue to the next node. The engine will again process any setup methods before executing the next node. This process will continue on each command node until either the command has completed successfully or any of the nodes fail during execution.

Once the command execution is complete, control of the command engine is returned to an idle state and the engine waits for the receipt of the next command. On receipt of the next command the sequence starts over again. The staging area is cleared and the next command is copied in from the activity node table.

4. Systems Engineering

Systems engineering has a major role in the development of the flight software command processing engine design described in the previous section. Since the systems engineering team designs the process flow for the command algorithms, they must work with the flight software developers to guarantee the command engine can perform all the required functions. Systems must also determine the design of the activity nodes themselves, which has a role in how the command engine is itself designed.

Systems must design the operational design concepts of how software is going to manage the system. They need to come up with the commands to be used, and what kind of command processing rules the software must follow. For example, if they require the software to handle multiple commands of different subsystems simultaneously then the command engine must be able to support multiple commanding threads. Systems engineering must also develop the requirements for interrupting a command in process, or even aborting commands, for other events such as fault management or mode changing events. Once all the requirements for the command handling system of the flight software have been developed, they must work closely with the software engineers to verify the design is adequate to operate the spacecraft commanding system.

The next task for systems engineering is to design the command activity nodes themselves. Once they develop the flow of the command algorithms they can now

create the activity node sequences. These sequences can be reviewed and tested against simulators and updates can be made and loaded without requiring new flight software builds. This will increase the design productivity of the command algorithms in the system and overall reduce cost and schedule of total system development. Systems will also need to develop a management control system of the activity nodes to record and manage the revisions used in the system.

The activity node design process can also include special activity node configurations needed to perform the defined tasks in different environments, such as integration and test in a 1-G environment. They may also find ways they may want to perform alternate algorithms on orbit for different mission tasks or mode functionality.

To make the development and management of the database a much more user friendly task, they may want to work with the ground systems group to develop a ground based tool to open, modify, upload, and read these activity node tables. This can be done at multiple levels and can even be done with a graphical tool developed in java or other graphical user interface language.

4.1. Ground based tools

Since the activity node software engine operates on a command database, the commands can be managed by a database management tool. Developing a tool allows

managing the flight software commands to be performed by engineers who have little if any software experience.

There are other users of the database tool. The integration and test program can make use of the tool to modify flight commands for use during satellite assembly and test.

The software algorithms are often not usable in a 1G ambient environment such as in a clean room. Sometimes flight software is used with only partially delivered electronics or subsystems. Having the capability on the floor to turn around updated software can be a big advantage. Another advantage to having a tool during system testing is to have the capability to modify the flight software for troubleshooting the satellite during testing.

The database management tool can also be delivered to the customer for managing the software for post launch operations. Adjustments to the command algorithms are often required once the satellite is on orbit. Adjustments are also often needed as satellites are in space over time, such as timing, voltages, and switching to redundant hardware. Having a single ground based tool can keep track of changes to the flight software database and keep the activity nodes in version control.

4.2. Activity Node Editor

The activity node editing tool developed on this project was a Java based program. It was developed to allow the user to open an existing database, modify the command

nodes, and export a new database for upload to the spacecraft. It was a graphical based tool that displayed the activity node of a command similar to Figure 3. The user could drag and drop activity nodes around and reconnect them in the desired order. When the user clicks on an activity node a second window pops up showing the details of all the parameters in the node. The user could modify these parameters and then save the changes. Once all the command changes were made, the tool could then automatically set the version of the table and build the upload file to be ready for the user to upload the new table. The table would then be tested before loading and using on the hardware, but this test would be much simpler than trying to validate and entire new version of flight software.

Operationally, there were several examples where the tool was beneficial to the success of the program. In one example, the flight software was delivered with a corrupted activity node in the default activity node database. Without the activity node editor tool, a full new flight software release would have been required. The tool allowed testing to continue with the existing flight software release. In another example, the sensor board in an electronics box was not installed. The activity node editor tool was used to remove the nodes that turned on, read, and verified the mechanism sensors and still kept the remaining functionality in the command to continue mechanism testing.

5. Conclusions

5.1. Advantages/Disadvantages

The activity node approach to developing flight software routines has many advantages. The sequences can be modified and uploaded independent of flight software releases. This is beneficial because updating flight software on orbit can be a difficult and time expensive task. Many times integration and test can be held up with flight software errors that shut down hardware testing. These cases often result in down time waiting for flight software to be fixed, tested, validated, and certified before being released to run on flight hardware again.

Another very important advantage is to the systems engineering group of the satellite program. The activity node design allows the implementation of the command algorithms to be developed directly by systems engineering instead of the software developers. This allows systems engineering to be more involved in the sequence design and more independent from the flight software team. Systems engineers often develop flow charts and sequence diagrams of how the command sequences should be implemented in the flight software. Flow charts and sequence diagrams transform easily into the activity node sequences.

Another group that can benefit from the activity node design of flight software is the integration and test team. This approach allows the ability to modify sequences for testing in the Integration and Test environment. For example, if routines need to be

modified to run in a high-bay ambient 1-G environment, they can be uploaded by the test team before operating the satellite in a high bay. When running in thermal vacuum test, they can select different routines that are correct for that test environment. They can also be modified to use redundant hardware or paths that may not be included in the default command algorithms. Launch and deployment sequences can be demonstrated with a modified activity node sequence that executes all steps of the launch sequence but skips the actual deployment of mechanisms and thruster firing.

Although there are many good features of running command sequences in a loadable table driven design, there are some disadvantages that need to be considered. The first one is the space satellite industry is very resistant to new ideas. Historically the industry likes flight software to be identical to what was executed in ground testing at the factory. Having the ability to upload new routines on the fly may be a scary idea. There may be concern that one bad load or a corrupted table would shutdown all command ability of the entire spacecraft. This is a real concern and should be considered when designing the flight software system.

There are some suggestions to minimize these risks. All new table updates should be tested and verified on the ground before loading to the spacecraft. The table could also be partially changed so changes to one command would not affect the other routines already on board. Another suggestion is to keep two versions of the table, one that can hold the previous version if something got corrupted in the new table. The other version may also be a default version that cannot be modified after launch so the

system will fall back to a known version of the routines. Checksums can be implemented to make sure all updates are loaded correctly and haven't been corrupted before executing. Also a good flight software engine design would validate all interface commands before sending them to the hardware.

If this approach is really of a great concern, the decision could be made to not implement the critical commands by this method. Since only a handful of commands could result in damage to the spacecraft these could be identified and implemented by other traditional methods. The activity nodes would still be very useful for all remaining commands used on orbit. The risk of losing or damaging the spacecraft due to activity node implementation really shouldn't be too much different than traditional flight software since protective measures and fault detection is required and operating regardless of which command method is chosen to be used.

5.2. Future Directions

The application of this solution in actual spacecraft is perhaps years away. Historically the satellite industry is resistant to change. Program management of satellites is for removed and not familiar with software and therefore resistant to trying 'new' ideas. The initial application of this proposed idea would probably be first applied to non-critical areas of flight software. One suggestion would be a payload or science instrument software application. This way if there are errors or unexpected results from

table loads the subsystem software can be reloaded and restarted. It would cost loss of data and time of recovering a payload but the spacecraft would not be lost. The idea may then grow to the main flight software subsystem but critical command and safe mode responses may be left in the traditional manner to again not cause loss of mission.

The concept of having a database driven software architecture may soon grow beyond command algorithms to other areas of software. These may include collection and scheduling of telemetry data to the ground, protective measures, fault detection and responses, or even the low level functions of the flight software itself. Even though consolidating all cases of flight software to be handled by a few different database engines may be a challenge, future programs may be able to find clever solutions to consolidate many parts of on-orbit software into this database driven design.

Once this type of software solution is learned and proven, future programs can take this knowledge of the software database engine design to future business. This may reduce the time and budget of software development on future programs. If the engine is powerful enough, it may be able to be reused with little change to the software.

The importance of the ground based database management tools will be key to how this can be developed in the future. The goal of the development of ground tools is they can handle complex database designs but still be manageable by systems engineering. This may be a good selling point to future programs using similar designs on new systems.

5.3. Conclusion

This activity node solution has been proposed as one way to include systems engineering in the design and implementation of the flight software. It takes a major function that was previously the responsibility of the flight software group and moves it to the systems engineering team. Since this design is independent of the flight software delivered binary releases, it can be managed and updated without minor or even major flight software changes. It can also be modified and used for different testing configurations during the integration and testing phase of satellite manufacturing.

This command processing design not only allows systems engineering to manage the way the software subsystem performs onboard command algorithms, it also reduces the complexity of the flight software. Many issues with software on spacecraft systems have been attributed to the growing complexity of the software designs. Improving this process is beneficial to the system as a whole and can help drive down development costs and schedules.

This type of system can also ease the process of flight software assurance. The command processing engine in the flight software is much smaller and less complex than a software system with hundreds of separate command algorithms in the final code. This greatly reduces the exercise of software acceptance testing because the engine can be validated, then the database elements can be tested separately. If elements of the command sequences are incorrect during testing, a new activity node

database can be changed on-the-fly and uploaded without a full software recompile and release.

Although this may show one method to help involve systems engineering in the flight software process, it still only deals with the commanding algorithm management of the software. Other areas of flight software may find a similar type approach to improve the process even further. The hope is this is a good start towards designing better spacecraft systems.

6. Bibliography

1. Blazing the Trail: The Early History of Spacecraft and Rocketry, Mike Gruntman, American Institute of Aeronautics and Astronautics, 2004
2. Future Directions in Flight Software Assurance, Elisabeth Nguyen, Robert Pettit, and Myron Hecht, Crosslink, The Aerospace Corporation, Spring 2011, <http://www.aero.org/publications/crosslink/spring2011/01.html>
3. NASA Study on Flight Software Complexity, Daniel L. Dvorak, NASA Office of Chief Engineer, 2009
4. Object-Oriented Design: A Responsibility-Driven Approach, R. Wirfs-Brock and B. Wilkerson, OOPSLA '89 Proceedings, October 1-6 1989, <http://delivery.acm.org/10.1145/80000/74885/p71-wirfs-brock.pdf>
5. The Role of Software in Spacecraft Accidents, Nancy G. Leveson, Aeronautics and Astronautics Department, Massachusetts Institute of Technology, <http://sunnyday.mit.edu/papers/jsr.pdf>
6. The Secret of Apollo, Systems Management in American and European Space Programs, Stephen B. Johnson, The John Hopkins University Press, 2002
7. Space Mission Analysis and Design, Second Edition, Wiley J. Larson and James R. Wertz, Microcosm, Inc and Kluwer Academic Publishers, 1992
8. Space Systems Failures, David M. Harland and Ralph D. Lorenz, Praxis Publishing, 2006
9. Spacecraft Systems Engineering, Peter W. Fortescue, John Stark, and Graham Swinerd, John Wiley and Sons, 2003
10. Standard Handbook for Aeronautical and Astronautical Engineers, Mark Davies (ed), McGraw-Hill Engineering 2003