****

**Command and Control System Automated Testing**

Rebecca McFadden

John F. Kennedy Space Center

Major: Computer Science

Fall 2020 Session

Date: 11 10 2020**Command and Control System Automated Testing**

Rebecca McFadden[[1]](#footnote-1)

Texas A&M University, College Station, Texas, 77840

# Nomenclature

API = Application Programming Interface

CSCI = Computer Software Configuration Item

IP = Internet Protocol

*KSC* = Kennedy Space Center

LCS *=* Launch Control System

*NASA* = National Aeronautics and Space Administration

SCCS = Spaceport Command and Control System

SCID = Software Configuration Identifier

SLS = Space Launch System

# Introduction

 To support the National Aeronautics and Space Administration’s (NASA) Space Launch System (SLS) rocket and the Orion capsule, designed to take humans back to the moon in 2024, Kennedy Space Center (KSC) has developed the Spaceport Command and Control System (SCCS) to monitor and control the launch. Within SCCS, the Launch Control System (LCS) is designed to allow console engineers to control and monitor the status of the launch and flight hardware, as well as issue commands to ground control systems and launch vehicles. The messaging software of LCS is responsible for handling the various data types that can be sent between the hardware and software components of the LCS. Since this system is interacting with numerous devices, controllers, and viewports in real time, the distribution of data across the system must be fast, but also reliable and accurate.

 To verify the accuracy and reliability of the system, developers on the project have created a set of tests to be performed that covers all operations allowed by the system. Given the extensive Application Programming Interface (API) provided by the messaging software, these unit tests are rather time-consuming and costly (in terms of man-hours) to perform. Therefore, an automated testing framework is used to perform supplemental tests automatically when updates are made to the code base.

# Objectives

My project was to extend and create new functional tests for the messaging software to cover a greater percentage of the code. Since this software is responsible for handling the communication layer between the different hardware and software components of LCS, my job was to make sure that all message types could be successfully sent or received accurately and in the correct format. The tests were based on existing manual test procedures run by the team. The following were my main objectives for this internship term:

* Gain an understanding of the messaging system and its functionality
* Investigate how the testing framework can operate and test the messaging system
* Increase the automated test code coverage for the messaging software

# Background

The messaging system is designed to provide reliable and secure communication between hardware and software layers of the LCS. It provides an API for access to the message bus, configuration and operational access to hardware during test and launch operations, and utilities for diagnostics, logging, and system data access. This level of abstraction obscures the lower-level interactions on the publish/subscribe software and makes it easy to use across other Computer Software Configuration Items (CSCIs). The messaging software offers this API on various Unix-based platforms, with high level programming language variants, since the various CSCIs use different operating systems or languages.

The messaging software uses the concepts of pre-defined “participants” to further abstract the communications across their system. Publisher participants can send data into the global space, where it can be read by subscriber participants. To reduce clutter and overlap in the global space and to avoid conflicts between different CSCIs publishers and subscribers, the messaging software uses the concept of partitions. A participant can be assigned to a specific partition and should only be able to read/write on that specific subdomain. Participants also help to define permissions and other settings. Initializing one of the messaging participants gives a CSCI that pre-determined level of access to the publish/subscribe software. The API provided by the messaging software also specifies message types for the various commands and associated data as well as the unique identifier info. By defining specific message types to be sent across the message bus, the various CSCIs can send and receive data in a standardized format and not worry about CSCI specific formatting.

Given the breadth of the message bus API and its diverse offerings in terms of operating systems and languages, the testing protocols for verifying the functionality of its services can be lengthy. A test driver program was created to allow engineers to run tests on a mockup of the message interface, but this still requires a human operator. To reduce defects without increasing time spent running manual tests, functional tests are employed. Instead of verifying functionality at a unit level or running system sequences, these tests use the automated testing framework to provide a layer of quality assurance between the two. The functional tests are executed as part of a nightly build, during regression testing by developers, and to verify new code is functioning properly before promotion into production. Functional tests are developed using an automated testing framework: a human-readable interpreted high-level programming language that executes instructions similarly to human executed test procedures. It can interact directly with the test driver script through a secure connection. While the automated functional tests do not eliminate the need for human testing, they increase the frequency that developers are able to test the code, which in turn reduces defects in the system.

# Approach

## Training and Setup

I spent the first few weeks of my tenure at NASA waiting for access to source code, development tools, and a computer I could use for developing software. While waiting for my developer accounts, I read through documentation on the software architecture and functionality, as well as completed trainings on the source control and security. After I gained access to the source code and a development box was set up, I familiarized myself with existing tests and sample code for the automated testing framework. I found a good amount of helpful documentation about this language on GitHub and I was able to use this documentation to help me better understand the tests that had already been written. To gain an understanding of the messaging software’s API, I used the test script to go through the various options for sending and receiving data. I was able to use a provided list of system-wide functional tests to experiment with the functionality of the test script and familiarize myself with the types of tests I would be writing.

## Verifying Authorization of Message Bus Participants

The message bus API uses the concept of participants to control the flow of information on the data bus. Participants are initialized for different CSCIs depending on the data they need to access or send. With this test suite, I began by verifying that the three main participants could be initialized on three different connections using multiple Unix-based operating systems. I then verified that all three could be successfully terminated and closed out cleanly.

The next phase of verifying proper initialization of the message bus was to verify authorization of participants. Since the API provides access to the data bus that goes between the LCS hardware and software CSCIs, it follows that they should maintain the security and integrity of this bus by managing the level of access for various participants. When a participant is initialized, the level of access is pulled from a database containing read and write access levels for each participant and topic. A different snapshot of the database is maintained for each updated version of the system software. Additional data necessary to support an operational event comprises an application layer.

Testing the authorization of participants is a multi-step process, as there are a few things that need to be verified. First, in order to publish or subscribe to topics on the data bus, a participant must have the correct read/write access as defined in the database. To test this, I made a copy of the database associated with the current application layer and modified it using pre-built automated testing framework libraries that allowed me to make database queries. I created a dummy participant, removed all read/write access for the participant, and then created a new entry for this participant that had no read/write access. I then verified that initialization of this participant failed due to a lack of read or write access.

I further testing this issue by modifying some of the initialization settings stored in a configuration file. These settings are used by the publish/subscribe software to configure how message traffic is controlled by participants for certain topics. This step of the testing procedure required modifying the configuration settings to set the default reader and writer to a dummy profile with the only setting being one relating to logs. Initially, I struggled with this part as I wasn’t sure I would be able to reproduce the step in the testing procedure with the automated testing framework libraries. However, I was able to find helpful documentation online regarding a parsing library for the automated testing framework and some programming language documentation that discussed ways to access and modify specific configuration elements. Once I modified this document in the test, I was able to verify that creating a generic publisher and subscriber failed if the required settings were not initialized in the configuration file.

I followed a similar, but much simpler, process for verifying that initialization failed without the required LCS configuration file. I set the environment variable that points to the configuration file to use a dummy directory. I then verified that a message bus participant could not be initialized without a proper configuration file.

The final test in this suite involved testing whether the publishers and subscribers could be assigned to partitions successfully. As discussed above, a participant can be assigned to a specific partition, and should only be able to read/write on that specific subdomain. For this segment, four participants were initialized (two publishers and two subscribers), with two different partitions. Various combinations of messages were sent by the two publishers and then the output on the subscriber terminals was compared to its corresponding publisher to verify that the pairs of publishers and subscribers on the same domain had the same messages.

## Verifying Send and Receive Operations for Data Types

This suite of functional tests was designed to test the send and receive operations for all message types across the different languages and Unix distributions supported by the message bus API. Before writing the automated tests for this suite, I ran each test manually to verify my environment as well as get a feel for the desired output. Typically, to run this test procedure, a software engineer would have to open five separate secure connections, one for each participant and two for viewing the logs. With the automated testing framework, I was able to use an extension of the built-in secure shell libraries to initialize these connections. This extension also helped me source the necessary environment variables from a pre-defined script written by the LCS team. This script helps to ensure the environment for the functional tests is clean, standardized, and mimics the environment of the launch hardware. Once the environment was set up, I initialized three participants on three different connections: a publisher and subscriber on one Unix-based operating system and a subscriber on a different Unix-based operating system. The automated testing framework provides a management system for secure connections, so switching between the three connections was simple.

I was then able to use the instances of the test script to verify that all data types could be sent by the publisher on one Unix-based operating system and received by the subscriber on another Unix-based operating system regardless of what high level language the subscriber was written in. To do so, I first verified that reading, with and without a set timeout, failed when the message bus was empty. I was able to use built-in automated testing framework keywords to verify that “Failed” and the correct reason appeared in the terminal output. Once I was sure the message bus was clear and that the timeout system was functioning correctly, I could publish a few messages and verify the correct unique identifier output on the subscribers. To test the timeout functionality, I prompted the subscribers to start reading both before and after a message was sent. I also tested to verify that the permissions for both participants were correct by attempting to send from a subscriber and read from a publisher.

## Testing Structured Data Publish and Receive Operations

Another format of data that the messaging software supports is Structured Data. Structured Data consists of measurement unique identifiers that are grouped into Structured Data Sets. The mapping of these to data sets can be found in the database for the given system software and application layer. Measurements that are grouped into a set can be published simultaneously (or time-homogeneously) and therefore can be read as a unit. The messaging software’s API provides functions for reading/writing structured data as well as verifying the unique identifier in a dataset. To verify the functionality of this section of the API, I first initialized a publisher and a subscriber participant and verified that I could send and receive any number of random data sets. I then queried the database for the current application layer to find a specific id in a dataset, and the id of an item not in a dataset. With these values, I tested the API for verifying if a given unique id is in a dataset or not and the API for finding the parent dataset id of a specific item. Finally, I tested the API to verify that publishing a large number of datasets resulted in the subscriber being able to read the data as a “set” instead of individual measurements.

# Conclusion

Functional testing is an important part of the software development process. It ensures not only a bug-free and high-quality product, but also that the overall “function” of a product works as advertised to a user. In the case of the messaging software, it is vital that the API the messaging software provides maintains its correct functionality even as improvements are made to other sections. All CSCIs rely on the messaging software to communicate their data to and from the various hardware and software components of the LCS. Testing is not a trivial task however, and can cost many man hours, which may reduce the frequency that a software project is tested. By automating the functional testing, the code can be tested thoroughly with every update. With the increased code coverage, I have provided for the messaging software, they are able to be more confident in their product and validate the success of their API much more frequently and efficiently.

# Acknowledgments

I would like to thank my mentors Jill Giles and Stephen Aument for welcoming me to the LCS team and providing me with guidance and support throughout my tenure at NASA. I would also like to thank my technical mentors Ken Brown and Leroy Richards for their shared expertise in terms of functional test creation and their assistance in getting my environment set up and debugging any technical issues I encountered. I wouldn’t have been able to overcome my challenges with the environment setup, tool startup, and network connections without their help. Jamie Szafran, Dorothy Zoledziowska, Mike Owens, and Tony Ciavarella were equally instrumental in debugging technical issues, providing feedback to my work through code reviews, and helping me with problems I experienced using the automated testing framework. I would like to thank Gwen Gamble and Kathleen Wilcox with the KSC Education Office for the incredible work they’ve done in terms of making this virtual intern term an educational and exciting experience. Finally, I would like to thank NASA for continuing to provide an impactful intern program despite the global pandemic.

# References

1Various, “Common Services (CMS) Computer Software Configuration Item (CSCI) Programmer’s Guide” NASA K0000175095-GEN, April 2019.

1. Software Development Intern, NE-XS, John F. Kennedy Space Center, Texas A&M University [↑](#footnote-ref-1)