SSC21-WKV-02

# Model-Based Systems Engineering and F': Proof of Concept Via the Creation of an On-Orbit Textual Command Parsing Component for the ABEX Mission

Michael Halvorson
University of Alabama in Huntsville, Complex Systems Integration Lab
Wernher von Braun Research Hall, 301 Sparkman Drive, Huntsville, AL 35899; 334-300-8131
MCH0043@uah.edu

CJ Short, Austin Bush, Brandon Scruggs, Joshua Lazenby, Sarah Kilgore, Sam Spearman, William Garrison, Paul Poe
Auburn University
Shelby Center, 357 W Magnolia Ave, Auburn, AL 36832; 334-401-0650
CLS0095@auburn.edu

## ABSTRACT

The Alabama Burst Energetics eXplorer (ABEX) mission is defining spacecraft architecture, behavior, mission phases, operational states, risks, and requirements in a Model-Based Systems Engineering (MBSE) Integrated Systems Model (ISM) using SysML in Cameo Enterprise Architecture (CEA). The satellite structural design can be exported from CEA as Extensible Markup Language (XML) specifications and imported to F', an open-source Flight Software (FSW) framework from NASA's Jet Propulsion Laboratory. F' contains background components intended to be connected to user-defined components in the XML after it is exported from the ISM; in this work, ABEX is representing F' background components in SysML Internal Block Diagrams from which the XML is generated. As a proof of concept for this MBSE-centric FSW implementation, the ABEX FSW team has created a Command Reader component from MBSE-generated XML and tested command enaction on a Raspberry Pi breadboard system for three test cases representing on-orbit command triggers.

## INTRODUCTION

Satellite Flight Software (FSW) systems maintain spacecraft functionality, enact predefined operations, and monitor status parameters such as temperatures, altitudes, radiation tolerance metrics, and subsystem interaction checks. FSW can be enacted by new frameworks with no prior development, such as the Apollo missions[1], redesigns of pre-existing software, such as the Ariane 5[2], or modular systems specific to a given satellite architecture, such as F' (pronounced F Prime). Early space system software engineers employed the first two approaches until the first modular frameworks were developed in the late 1990s and early 2000s[3].

Systems theory in FSW explores object-based relations and ontologies[4], which was the basis for the Apollo program's FSW development[1]. Rooting FSW maturation in Systems Engineering (SE) principles provides flexibility and consistency throughout spacecraft mission phases; repurposing FSW developed with this approach can reduce program development time compared to building new software. However, the redesign of pre-existing systems can result in unanticipated errors or catastrophic failures with previous methodologies not translating to new components exactly[2]. This does not mean that reuse should be avoided entirely; myriad NASA missions have included at least some repurposed or legacy code from previous missions[5]. Risk analysis and criticality matrix definition must be performed when reusing software, as with any subsystem, to avoid potential mission-ending errors.

The use of FSW frameworks with pre-built or modular components are convenient when designing flight software for a new mission, especially for smaller missions such as CubeSats. Modular approaches such as KubOS[6], core Flight System (cFS)[7], and F'[8] provide basic FSW frameworks and tools for developers to utilize or create subsystem functionality for a mission. KubOS implements components via user-level programs which can be run as one-off executions or continuous processes and employs Graphical User Interface (GUI)-based interactions for integration and testing simplicity[6]. Users can only add predefined components to their satellite based solely on what KubOS provides. This approach becomes cumbersome if users want to implement a technology demonstration or science instrumentation component for their project. cFS from NASA Goddard provides a similar development framework and basic, reusable

components which can be pieced together for new applications and allow for easier reconfiguration of the topology at runtime[7]. Neither KubOS nor cFS represent solutions intended to operate in conjunction with SE artifacts.

F' is an open-source FSW solution created by the NASA Jet Propulsion Lab (JPL) that can build spacecraft architectures directly from Model-Based Systems Engineering (MBSE) products, making it ideal for small missions such as CubeSats and SmallSats. It has also been used to operate the Mars Drone Helicopter Mission: Ingenuity[9]. F' provides an autocoder framework for commands, telemetry, and events, standard operation modules such as uplink, downlink, and command sequencing, and peripheral support including abstractions, modeling, testing, and basic ground station compatibility. The main advantages in using F' are standard components, abstractions, modularity, reusability, efficiency, scalable features, and the unit testing framework. The architecture of F' consists of components, which are the framework building blocks and represent system structures or behaviors, ports for communication between components, and topologies for a top-level system layout[10]. A motivation for using F' on the Alabama Burst Energetics eXplorer (ABEX) mission is its inherent compatibility with Systems Modeling Language (SysML)-defined structures that can be modeled in Cameo Enterprise Architecture (CEA). CEA, an MBSE suite, can be used to model all aspects of the spacecraft using SysML diagrams. A MagicDraw plugin exports XML files from SysML Internal Block Diagrams (IBD) representing the spacecraft's components, ports, and topologies, and F' imports the XML files and their respective topologies to create operative C++ and header files for the system. The MBSE-defined architecture can also be used to define functional flows as SysML Activity Diagrams and command operations as Sequence Diagrams with increasing levels of complexity, meaning hardware organizations, Concepts of Operations, and satellite FSW can be generated from the same MBSE platform.

The present work is twofold:

1. Prove that reusable components internal to F' can be modeled as IBD components in CEA, successfully connected to spacecraft components in the IBD, and exported to XML resulting in a method to represent internal F' functionality in MBSE.

2. Use this F' MBSE proof of concept to create a new command structure that parses textual commands on-orbit instead of uploading bit level commands; this could be considered a reworked approach to implementing similar functionality as *tinyseqgen*.

While F' reusable components were modeled directly in CEA and connected to user-defined components in SysML IBDs, the Command Reader was created by augmenting XML files after the MagicDraw export.

The following sections demonstrate how F' was augmented for the ABEX mission to provide autonomous spacecraft functionality by running task networks[9] sequentially from an uploaded text file. This augmentation begins with the creation of a Command Reader component that was originally implemented as an "LEDControl" component. The LEDControl component was integrated with pre-existing F' components in CEA as a SysML IBD. XML files for the LEDControl component were exported to F' where C++ functionality was written to parse a given text file and execute the task network commands specified in the text file. Three test cases were performed using the Command Reader to demonstrate common functionality for spacecraft operations. Tests were chosen to represent the system's response to external stimuli, environmental triggers, safety cues, and command verification. All tests were successfully executed. The Command Reader component is an advancement towards MBSE-centric spacecraft design and autonomous operations capability but does not represent an improvement over existing F' command implementation. The purpose of this work is to step through established F' capabilities in a new way and prove representation of existing F' background components in SysML is possible, not define new or improved functionality in the FSW framework itself.

## MBSE IN FLIGHT SOFTWARE

The MBSE approach to creating modular FSW in F' utilizes component structures defined as IBDs in SysML, an extension of the Unified Modeling Language; pertinent diagrams for non-F' MBSE include Requirements Diagrams, Block Definition Diagrams, Activity Diagrams, Sequence Diagrams, Use Case Diagrams, and State Machine Diagrams. CEA generates XML files using the MagicDraw plugin based on user-defined components and assigned stereotypes. Stereotypes are a method of extending a Metaclass onto a given object; Metaclasses are how CEA understands when to generate an associated XML component and its corresponding relationships. Whenever a user assigns the Metaclass "class", the corresponding object can then be exported successfully. The Metaclass "class"

is associated with the "active," "passive," and "queued" component types in F'. Stereotypes can be applied to anything the user can edit. The built-in CEA diagram for these component implementations is depicted in Figure 1.

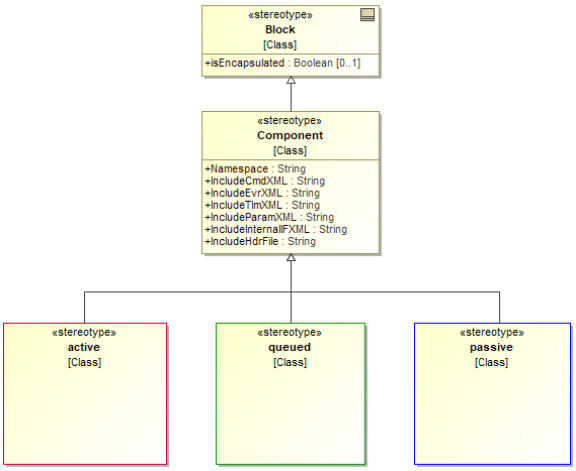


**Figure 1: Exportable Component Types**

Port types are defined within the "class" Metaclass. Ports have a set of Metaclasses so CEA can recognize the need to be exported but are implemented differently from components. Depicted in Figure 2 are the three F' input port types, shown in respective component type coloring, and a Cmd port. Figure 3 provides examples of pre-defined F' output ports.

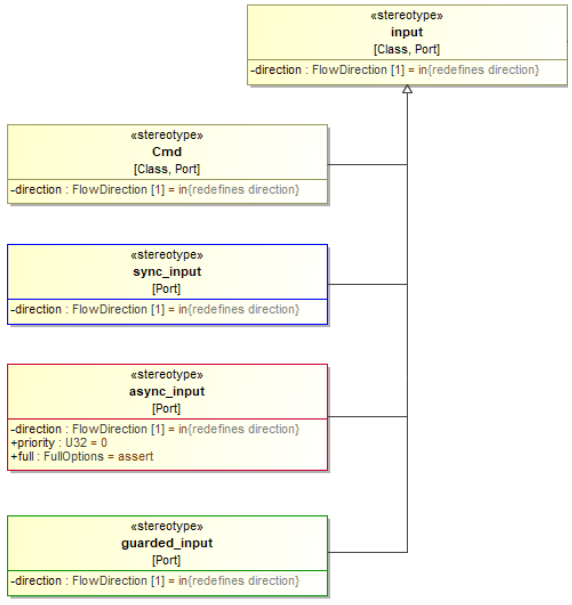


**Figure 2: Types of Input Ports**

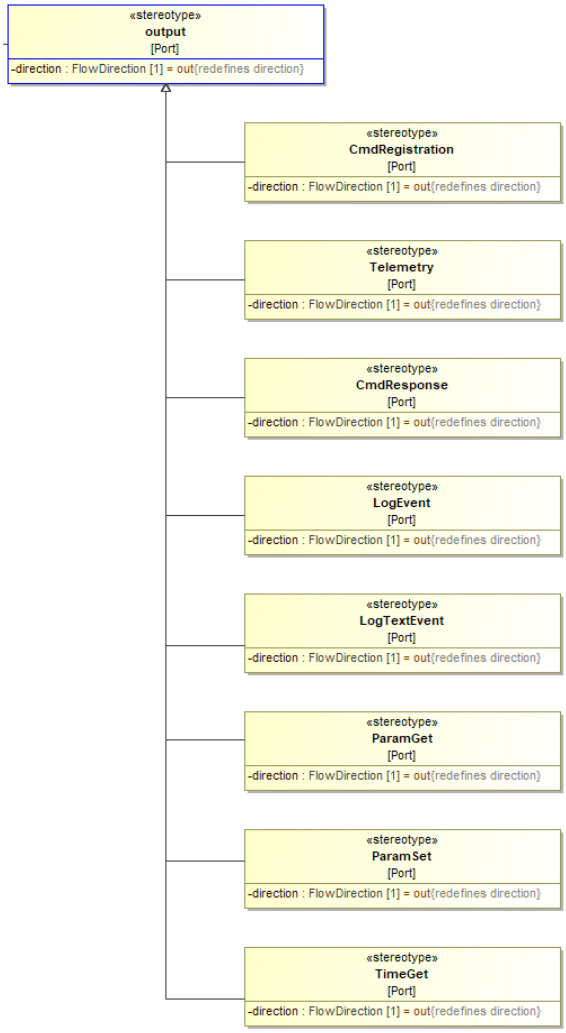


**Figure 3: Examples of Pre-Defined Output Ports**

Exportable stereotypes are color-coded, and these colors appear in the input ports. It is possible to give ports to components that do not fit this color coding, but users will typically run into errors that require adjustments such as giving a component a command to run despite it being a passive component. An overview of port types and uses is provided in *Bocchino et al*[9] but using F' ports in non-standard ways may result in non-standard error solutions.

If a component exists but does not have an IBD it can still be exported; all that is generated is an empty file containing no useful information. This is a result of not identifying relationships between other associated components. This also implies that CEA can export any kind of component diagram, but that does not make it useful. Sequence, Activity, and State Machine Diagrams are able to generate XML files but nothing of use to F' is in the XML. Because of the strict

Metaclass designation control over XML generation, the primary tools for effective creation and application of autogenerated XML files for use in F' are the components, attachable ports, and connections in IBDs.

XML-based satellite architecture exports and F' together represent a Technology Readiness Level 8 or higher FSW solution; the strategy was successfully used on the ASTERIA mission by JPL to detect exoplanets from Earth's orbit[11]. Information about specific ASTERIA subsystems or command designs in F' and CEA is not publicly available, a circumstance that partially motivates the present work. Motivation also stems from the convenience of creating new components and functionality using MBSE exports to F'. This process, illustrated in Figures 4 and 5, begins with an IBD design of the new component in CEA. F' autogenerates a C++ file from the IBD XML with implementation stubs and a corresponding header file. Implementation of desired component functionality is added in the new C++ and header files, and unit tests are generated and implemented. This process allows for integration of newly defined components. For the ABEX mission, text-based command capabilities were created using this process as a proof of concept for autonomous command structures.

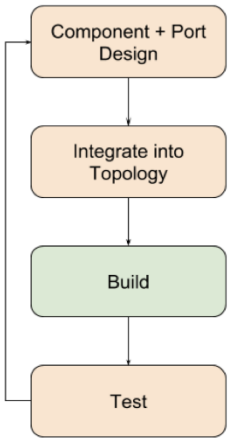


**Figure 4: F' Implementation Process[10]**

F' supports execution of individual commands which provides three options for commanding the satellite: commands manually uplinked by a user monitoring the satellite from a Ground Control Station, software state changes automated with commands implemented in C++ set to read from chronological or environmental stimuli, or sequential lists of commands pre-defined for execution as task networks. The optimal approach for FSW command autonomy via task networks is likely a hybrid of all three options.

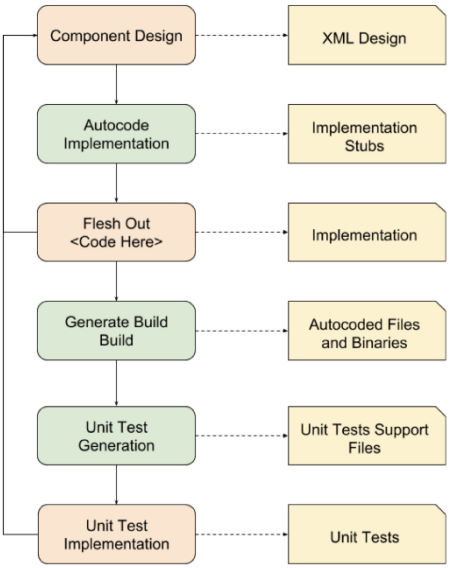


**Figure 5: Detailed F' Implementation Process[10]**

## F' COMMAND READER AUGMENTATION

Spacecraft functional autonomy is achieved by augmenting the F' command structure to sequentially read command task networks. F' uses a Command Sequencer to order commands given to F' in a file with a defined sequence for the commands. F' then sends the commands to Command Dispatcher to be executed in the specified order[12]. Information available about the file type and syntax provided to the Command Sequencer and where it is located in F' is limited; a link was provided in *Bocchino et al.*[9] but is no longer functional. The ABEX FSW team made the decision to create a new component that worked similarly to the Command Sequencer.

The Command Reader was not built directly in CEA; it was created by augmenting XML from an LEDControl topology created for test purposes, shown in Figure 6. This Command Reader component implementation strategy was somewhat backwards but ultimately successful.

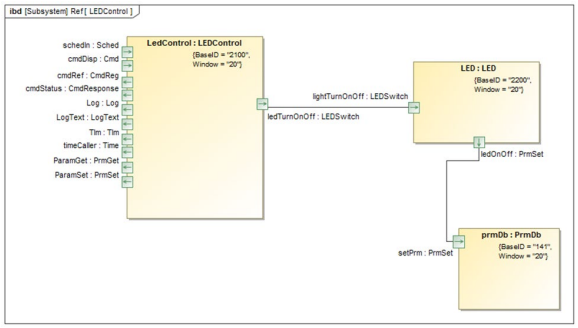


**Figure 6: LEDControl Subsystem IBD in CEA**

In practice, the data passed by the LEDControl component should be connected to PolyDb, not PrmDb[12]. The only differences between the LEDControl component and the Command Reader are that the Command Reader does not have either the LEDSwitch or the SchedIn ports pictured on LEDControl.

The Command Reader component reads and executes predefined commands from a file that can be uplinked from the ground station or compiled onto the spacecraft memory. Commands are input as a text file which includes the list of commands in the order they will be executed. In terms of task networks, only the command sequences are defined for this proof of concept, not the conditions or impacts[9]. There are three categories of commands that can be included in the command text file: F' Commands, C++ Commands, and Linux Commands. F' commands have three parts: the name of the component which executes the command, the name of the command, and the arguments passed into the command. The Command Reader must prefix text to every F' command listed in the text file to be able to run the command through the terminal using the built-in F' functionality for executing commands as text commands in the terminal. The Command Reader simulates this implementation by using the C++ `system` function from unistd.h to call the commands in a virtual terminal. Prefix text for every command:

```
"fprime-cli command-send -d
Top/RefTopologyAppDictionary.xml "
```

The C++ command category includes a 'wait' command. Wait uses the C++ *usleep* function from unistd.h. The format for wait is the word wait followed by the number of seconds the user wants to wait; if no time is specified, the wait time defaults to five seconds. The Linux command category includes a 'shutdown' command. The syntax for this command is the text "shutdown." This command uses the C++ `system` function to call the "shutdown now" Linux command in a virtual terminal.

Commands for actual ABEX subsystems will be created by the FSW team based on SysML Activity Diagrams and Sequence Diagrams. ABEX subsystem teams from around the state will build the functional flow of subsystems in Activity Diagrams which also serve to identify interface requirements. The FSW team will use the provided Activity Diagrams to create Sequence Diagrams which can subsequently be used to define commands and task networks for the functional flow.

An important aspect of the Command Reader is to be able to log which commands are being executed and if they are successful. Using a simple print statement in C++ fails to log these messages at the correct time because the print statements in C++ operate solely within the boundaries of the terminal and do not have inherent compatibility with F'. The messages will either be delayed in their output or will not appear at all depending on where the print statement is located within the C++ command. The `logMsg` method in the Logger class of Fw/Logger is used to write to the F' logs. The developer will need to import the Logger.cpp file within the C++ file that needs logging functionality:

```
#include <Fw/Logger/Logger.hpp>
```

The developer can now write to the logs in real time:

```
Fw::Logger::logMsg("Log Message");
```

This method works similarly to a typical C++ print statement except it prints directly to the logs in the F' GUI and prevents any delay from occurring when writing to the logs.

## TEST METHODOLOGY

Three tests were conducted to ensure the implementation of MBSE-defined subsystems and the use of F' is suitable for creating FSW. Each test represents a required functionality for spacecraft operations. Tests were chosen to represent the system's response to external stimuli, environmental triggers, safety cues, and commands. Three I/O devices were used to represent these functionalities: an LED representing an indicator for confirmed recognition of external stimuli, a button representing environmental triggers or uplinked commands, and a temperature sensor representing an input device to be polled like an altitude sensor, star tracker, or inertial measurement unit. Additional components such as LEDControl, ThermalControl, and Thermometer representing subsystems and hardware were also created as IBDs in CEA to conduct these tests. These components included functionality that controls and interacts with hardware components and commands that control parameter values.

All tests were conducted on a Raspberry Pi 4 Model B running Ubuntu 20.10 desktop and the latest version of F'. Additional hardware was controlled through general purpose I/O pins on the Raspberry Pi. A TMP36 temperature sensor was used to read temperature, and an MCP3008 analog to digital converter was used to convert the analog signal from the TMP36 to a digital signal for processing on the

Raspberry Pi. An overview of the three tests were defined:

1. Initialize the system, blink for 10 seconds at any rate, and turn off the board. Turn back on, blink faster for 10 more seconds, and turn off the board. Basic command response is represented.

2. Initialize the system, turn on the board, and blink until the button is pressed. Wait 5 seconds and turn off the board. Response to external stimuli and wait functionality are represented.

3. Initialize the system and turn on the temperature sensor in ThermalControl. When the button is pressed, stop taking data from the temperature sensor. When the temperature exceeds a thermal set point, enter Safety. Loop functionality and system state transitions due to environmental triggers are represented.

For the tests to successfully pass, the On-Board Computer (OBC) had to initialize itself, start F', and run a sequence of commands autonomously. The only input needed was a button press for the second and third tests. The goal was to simulate space operation scenarios since the satellite ideally operates autonomously except for minimal input from the Mission Operations Center. Sequence Diagrams for each test and can be seen in Figures 7, 8, and 9. Future Sequence Diagram iterations will feature criteria for what aspect of F' is being connected rather than abstracting a process to F' in general.

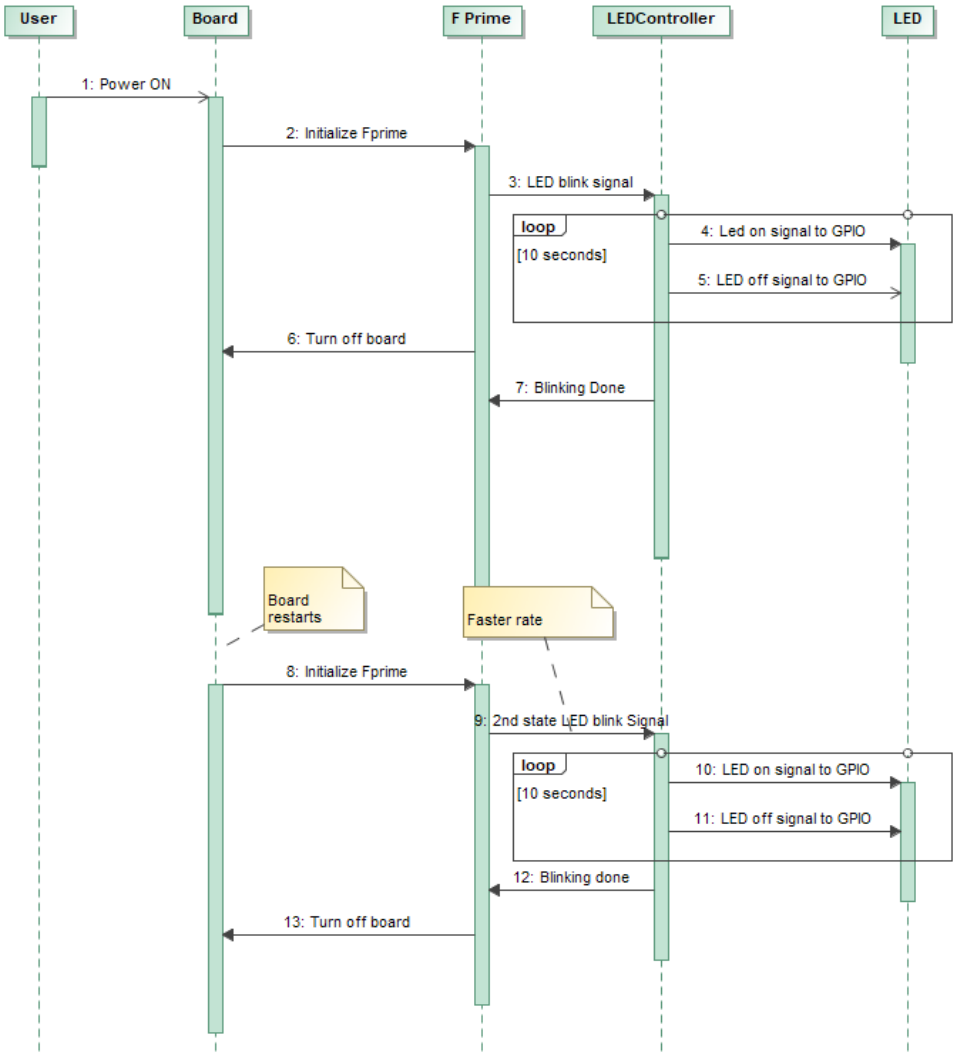


**Figure 7: Test One Sequence Diagram**

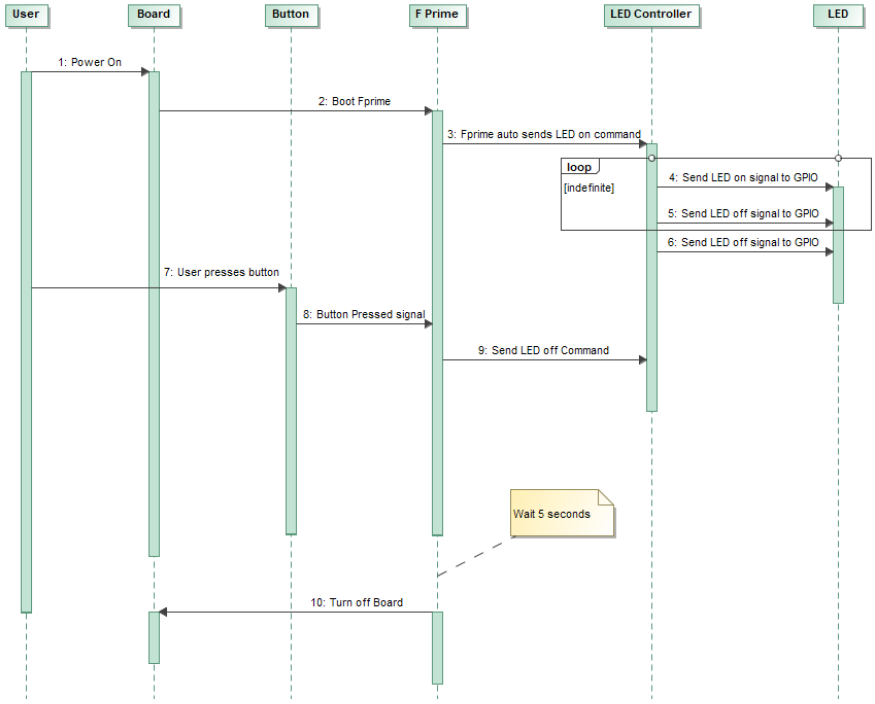


**Figure 8: Test Two Sequence Diagram**

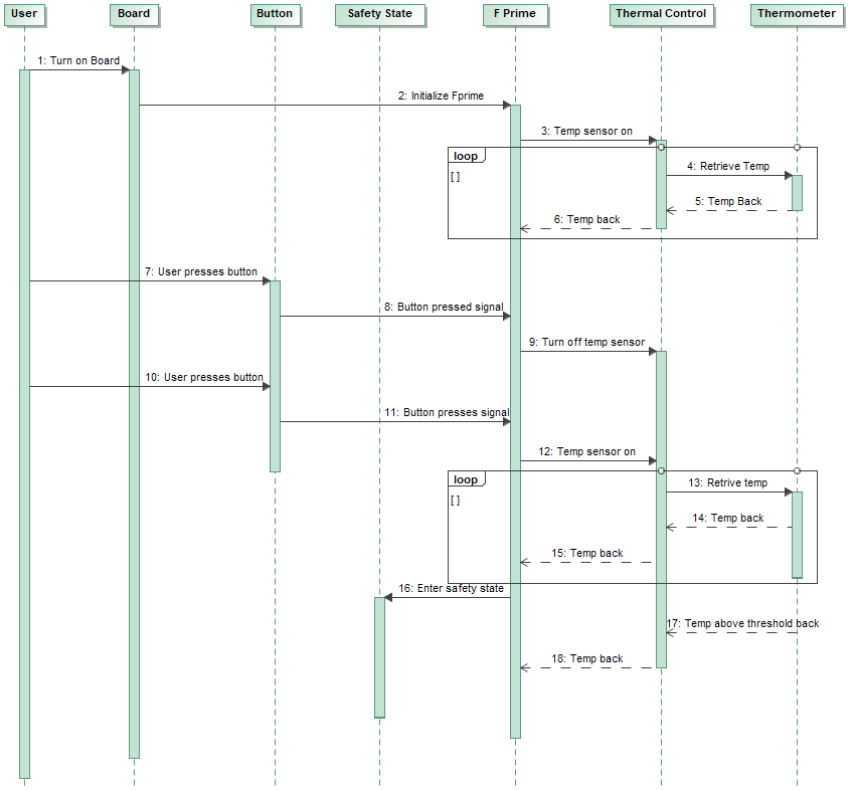


**Figure 9: Test Three Sequence Diagram**

The wiring diagram for all three tests can be seen in Figure 10.

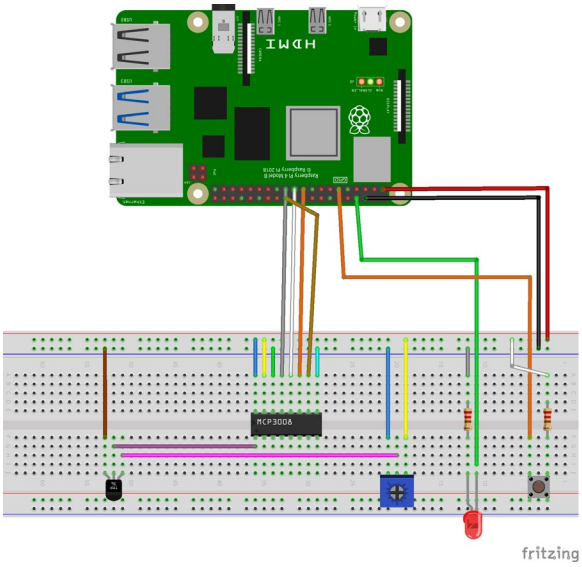


**Figure 10: Wiring Diagram**

## RESULTS

All three tests were successfully run. The LED blinked for 10 seconds and then blinked faster for 10 more seconds in test one. The system was unable to turn itself off and back on as desired because Raspberry Pi does not support *rtcwake*, so this part of test one was marked as a failure. The LED blinked until the button was pressed, and then the board turned off for test two. The button turned the thermometer on and off for test three, and when the temperature of the thermometer passed a threshold, the system entered a safety state. Below is the list of commands executed for test one.

```
LEDControl.START_LED -args 2
wait 10
LEDControl.LED_ON_PRM_SET -args 0
wait 2
LEDControl.START_LED -args 10
wait 10
LEDControl.LED_ON_PRM_SET -args 0
```

The log output for test one is shown in Figure 11.

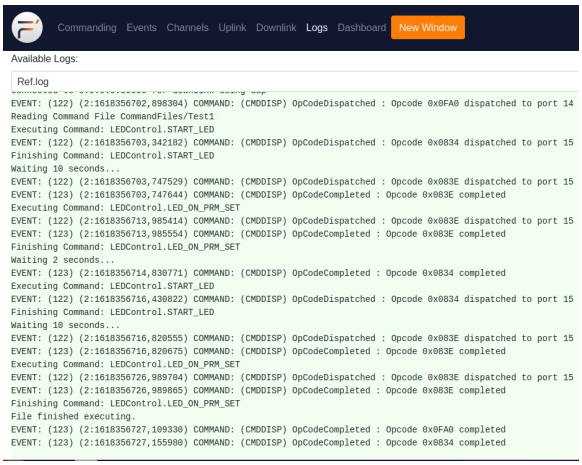


**Figure 11: Test One Log**

Each command was executed sequentially. The command to turn the LED off is the built-in command to set the parameter LED_ON in the parameter database. This parameter was set to one when the LED was turned on and set to zero to turn the LED off. The output for this test shows which commands are executed and when. The lines which state more clearly the name of the command and what it was doing were added for this project. The log showed when each command began and ended, the wait command with the given number of seconds, and when the entire command file finished execution.

The first command file input for the second test was much simpler since most of the functionality came from the button as seen below.

```
LEDControl.START_LED -args 10
```

The only command being run initially for this test was to turn the LED on. All functionality outside of the LED beginning to blink was done when the button was pressed, and another file was read.

```
LEDControl.LED_ON_PRM_SET -args 0
wait 5
shutdown
```

The log output of both command files is shown in Figure 12.

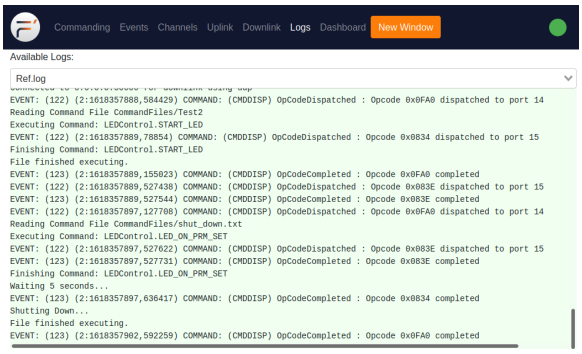


**Figure 12: Test Two Log**

The third and final test had more distinct parts than the first two tests, but the command file only initialized the thermometer peripheral to read a temperature in ThermalControl.

```
thermometer.IS_READING_PRM_SET -args
1
```

Three other command files were also used for this test: two to turn the thermometer on and off and one to represent a safety state.

```
thermometer.IS_READING_PRM_SET -args
0
button.IS_PRESSED_PRM_SET -args 1
```

The button command is used to prevent registering multiple button presses from holding the button down for too long. The commands to turn the thermometer on are similar.

```
thermometer.IS_READING_PRM_SET -args
1
button.IS_PRESSED_PRM_SET -args 1
```

The command file for the safety state does not represent any actual safety functionality but is used as a proof of concept that external stimuli can trigger the execution of a command file. The safety state in this instance only turns on the LED.

```
LEDControl.START_LED -args 2
```

This test proved that F' can operate with an external device over time. The system can also stop and start data flow and move into other subsystems based on input data from the temperature sensor in this case.

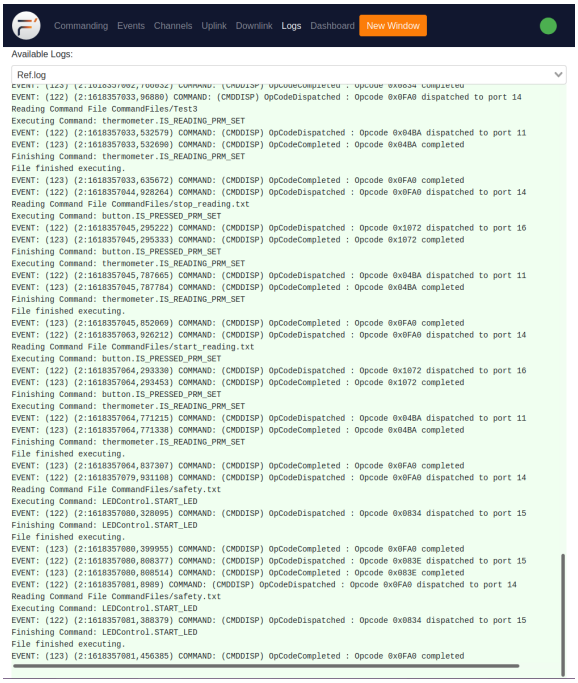


**Figure 13: Test Three Log**

## FUTURE WORK

With the ability to generate new components in CEA as SysML IBDs and connect them to similarly represented, pre-existing F' components proven viable, ABEX will now begin to model the entire spacecraft architecture, behavior, mission phases, operational states, risks, and requirements in an MBSE-centric ISM. From a central authority, ABEX will be able to generate FSW, requirements verification artifacts, risk criticality matrices per subsystem, science traceability matrices, education traceability matrices, Concepts of Operation, and Day In The Life (DITL) test procedures. In the near future, the software-from-MBSE approach may be considered standard practice among professional space systems developers.

## CONCLUSIONS

FSW is difficult to create and maintain, and various frameworks such as F', KubOS, and cFS have been created to simplify the process. These frameworks typically allow developers to spend more time implementing specific operations by including general functionality that most satellites will require. Implementing new components that do not exist as options in pre-existing FSW frameworks are most easily created using an MBSE approach that naturally exports to FSW.

The ABEX mission uses F', an open-source framework created by NASA's JPL for use with small-

scale satellites. SysML IBDs are created in CEA, and the MagicDraw plugin allows for auto-generated XML files to be exported into F'. F' generates C++ files from the XML files received from CEA to allow implementation of user-defined satellite functionality.

The common approach to using F' was to export user-defined IBDs as XML and connect them to background F' components after the export. In this work, the ABEX FSW team included background F' components in the IBDs, created a user-defined component to read in textual commands in a new format, connected the Command Reader to the background F' components in the IBD, exported functional XML in a single package, and tested Command Reader functionality on a Raspberry Pi. Three tests representing on-orbit command-initiating events were successfully completed. The ABEX team here created the Command Reader from exported XML defining an LEDControl component, but the structure of LEDControl and Command Reader were almost identical. This leads ABEX to believe new components can be directly created as IBDs that not only define spacecraft functionality but also alter the functionality of F' in general. MBSE-implemented F' does can do more than simply include F' background components, it can change how F' operates fundamentally. This should be viewed as yet another type of versatility that this powerful FSW framework provides.

***Acknowledgments***

The ABEX program would like to thank Dr. Xiao Qin of Auburn university for his continued leadership, guidance, and expertise in satellite FSW development. The ABEX program would also like to thank the Alabama Space Grant Consortium for its dedication to bringing space system design and fabrication opportunities to hundreds of engineering students across the state of Alabama, specifically Dr. L. Dale Thomas, Debora Nielson, and Brooke Graham. ABEX thanks the engineers at JPL who have helped undergraduate students learn their incredible software, especially Jeff Levinson, Michael Starch, Tim Canham, Rob Bocchino, and Leonard Reder. ABEX thanks Trent Rich, who created a manual for the MagicDraw to F' plugin interface and assisted the students in utilizing it. Finally, ABEX thanks the undergraduate software students who came before the present team and built the foundations for consistently improved work.

***References***

1. Hamilton, Margaret H., and William R. Hackler. "Universal systems language: lessons learned from Apollo." *Computer* 41.12 (2008): 34-43.

2. Orrego, Andres S., and Gregory E. Mundy. "A study of software reuse in NASA legacy systems." *Innovations in Systems and Software Engineering* 3.3 (2007): 167-180.

3. Bätz, Bastian, "Design and Implementation of a Framework for Spacecraft Flight Software," Institute of Space Systems, University of Stuttgart, 2020.

4. Whitchurch, Gail G., and Larry L. Constantine. "Systems theory." *Sourcebook of family theories and methods*. Springer, Boston, MA, 2009. 325-355.

5. Malcom, H. and Harry K. Utterback. "Flight Software in the Space Department: A Look at the Past and a View Toward the Future." Johns Hopkins Apl Technical Digest 20 (1999): 522-532.

6. Kubos (2018) KubOS https://docs.kubos.com/1.5.0/index.html.

7. NASA July 10, 2020 Lisa Kane https://cfs.gsfc.nasa.gov/Features.html

8. F Prime: A Flight Proven, Multi-Platform, Open-Source Flight Software Framework (2017) https://github.com/nasa/fprime.

9. Bocchino, Robert, et al. "F Prime: an open-source framework for small-scale flight software systems." (2018).

10. Starch, Michael. "F Prime: NASA Open Source Meets Small-Scale Flight Software." (2019).

11. Bocchino, Robert, Brian Campuzano, and Len Day. "Applying the F Prime flight software framework to the ASTERIA CubeSat." (2018).

12. The Discerning User's Guide to F' https://nasa.github.io/fprime/UsersGuide/guide.html