OpenSatKit User's Guide

Version 1.4 , Released August x, 2018

Table of Contents

1.0 Introduction

OpenSatKit was developed for the following reasons:

1. **Serve as a distribution of the core Flight System (cFS).** The cFS is an open architecture that has separately configuration managed components. NASA controls the framework and Application Programming Interfaces (APIs) and the cFS community including NASA provides components.
2. **Serve as a cFS educational platform.** This includes teaching users about the cFS itself and also providing an environment that could be used as a classroom "lab" for doing exercises.
3. **Provide an application prototyping environment.** It supports creating, integrating, and removing apps from the kit. It can also be used for initial configuration of apps for a mission.
4. **Support target embedded platform evaluation and initial ports**. A benchmarking app allows a user to run and compare benchmarks for different platforms. A plan is to create a platform verification app that will verify a target platforms functional behavior.

The kit is not intended to support a mission throughout the entire lifecycle. For example it does not support hardware in the loop configurations. This does leave a gap if a mission is using COSMOS as its ground system for spacecraft integration and test and/or mission ops. The current options are for a user to create COSMOS extensions to the kit to meet their mission needs or to port the portions of the kit (command & telemetry definitions, pages, scripts, etc.) from the kit to the COSMOS platform that has been configured/extended for their mission.

This section describes the motivation for creating the starter kit and a high level introduction to each of the architectural components. If you want to jump in and start using the kit go to Section 2. If you want to use the kit to customize the Core Flight System (cFS) sections 3 and 4 explain how to manage applications and transition from the kit virtual machine to your target hardware platform, respectively. Please keep in mind the starter kit is composed of three complex products and the goal of this documentation is not to explain the details of each product. This document is written from the perspective of a flight software (FSW) developer that wants to use the cFS to control an embedded device.

Section 5 describes how to maintain the kit and delves into the starter kit's design. A typical product user's guide wouldn't include this information, but just as the kit's components are 'open architectures' the kit itself is open and may be modified by the user to be part of an operational system so this information will be helpful to those users.

1.1 Motivation

Until around 2010 the development of spacecraft flight software (FSW) was performed by large organizations that had custom proprietary solutions. FSW ran on processors lagging terrestrial processor performance by orders of magnitude often forcing software engineers to opt for performance over generalized solutions. With the exception of communications satellites many organizations produced 'one off' customized satellites to designed for a specific application. These product development lifecycles were often 5-10 years so even when FSW reuse was occurring the maturation of the reusable artifacts across products was very slow.

During this period from roughly 2005 until 2015, the National Aeronautics and Space Administration (NASA) Goddard Space Flight Center (GSFC) developed the Core Flight System (cFS) and in January 2015 the cFS was released as open source. This was great news for the aerospace community, however there were many challenges with people actually adopting the cFS for their missions. The cFS is a reusable FSW architecture that provides a portable and extendable platform with a product line deployment model. As an open architecture, the cFS can be technically challenging for new users to configure and deploy. In addition, as a government organization, it is difficult for NASA to implement an open source product business model.

This starter kit addresses these issues by providing a fully functioning flight-ground system that runs on a desktop computer. The starter kit components are shown in Figure 1. Ball Aerospace's COSMOS, a user interface for command and control of embedded systems, is used as the ground system. The cFS running on Linux provides a desktop FSW component. The 42 Simulator provides a simulation of spacecraft attitude and orbit dynamics and control. See Appendix B for details on obtaining more information on each of these components.

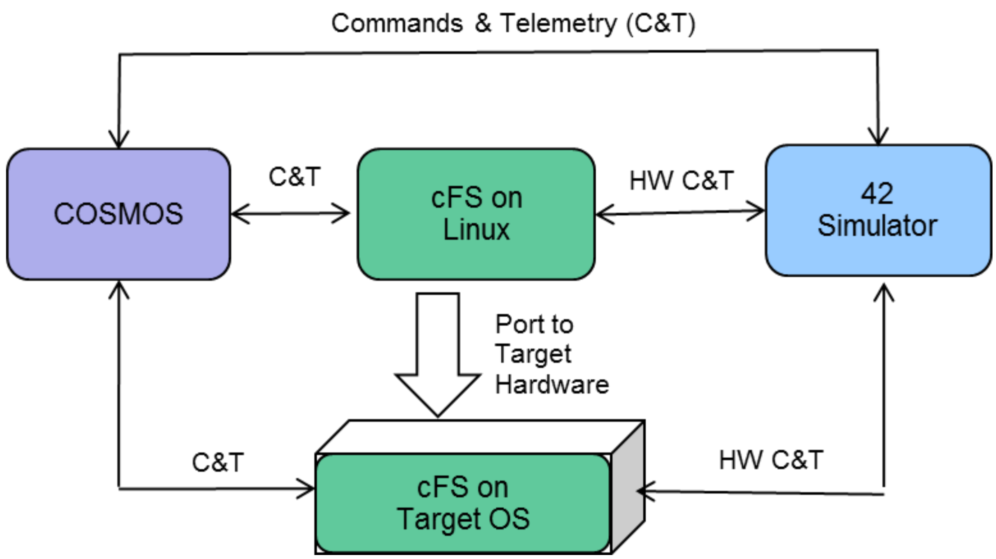


Figure 1-1 – Starter Kit Block Diagram

Starting with an operational flight-ground system makes the FSW developer's job much easier. They can focus on porting the cFS to their target platform, tailoring the kit's cFS components to their needs, and adding new mission-specific components. A future version of the kit will include a low cost commercially available target. This version describes the steps necessary for using the kit to verify that the cFS with the kit apps has successfully been ported to any target.

The starter kit also serves as a cFS training platform. It provides demonstrations to highlight common cFS features and it contains a tool for automatically creating a "Hello World" application. Since it is freely available and easy to install, it can be used as a platform for academic projects.

The cFS can have a significant impact on a mission's FSW costs. The cFS has provided about a third of the FSW on recent NASA missions using source lines of code (SLOC) as a metric and excluding the operating system from the SLOC count. Much of the functionality provided by the cFS is based on decades of FSW experience. This functionality can be very beneficial to inexperienced teams because they may not even recognize that they may need some of the functionality provided by the cFS, especially the inflight diagnostic and maintenance features.

The remainder of the introduction provides a brief description of the cFS, COSMOS, and the kit's architecture. If you are familiar with these components you can skip to Section 2 to get started with using the kit.

1.2 cFS Overview

Before jumping into the cFS architecture it's worth understanding some of the rationale behind the design. Prior to the cFS NASA GSFC FSW reuse efforts had limited success in reducing cost and schedules. Early reuse efforts used a "clone and own" approach where a new project would copy FSW components from one or more previous missions based on functional requirement similarities. This informal source-code based approach to reuse proved difficult for managers to control the scope of the changes and as a result a comprehensive verification and validation effort had to be performed for the new mission which severely limited the cost savings. In addition since FSW components were not configuration managed independent of projects, component quality did not necessarily increase because a single lineage for each component was not maintained.

To address these challenges the Goddard's FSW Branch formed a team of senior engineers to perform a structured heritage analysis across a decade of missions. The initial funding was from non-mission sources which allowed the engineers to participate uninhibited by near-term mission schedules. The diversity of the heritage missions (single string vs. redundant string, varying orbits, different operational communication scenarios, etc.) provided valuable insights into what drove FSW commonality and variability across different missions. The team took the entire FSW life-cycle into consideration, including in-orbit FSW sustaining engineering, as they performed their analysis. Identifying system and application level variation points to address the range and scope of the flight systems domain. The goal was to enable portability across embedded computing platforms and to implement different end-user functional needs without the need to modify the source code. The cFS uses compile-time configuration parameters to implement the variation points. Figure 1-2 shows the results using a classic software engineering "V-model". The shaded components are cFS artifacts and the *<p>* notation indicates a parameterized artifact. This lifecycle product line approach dramatically increased the number of reusable artifacts and changed how future missions would approach their FSW development efforts.

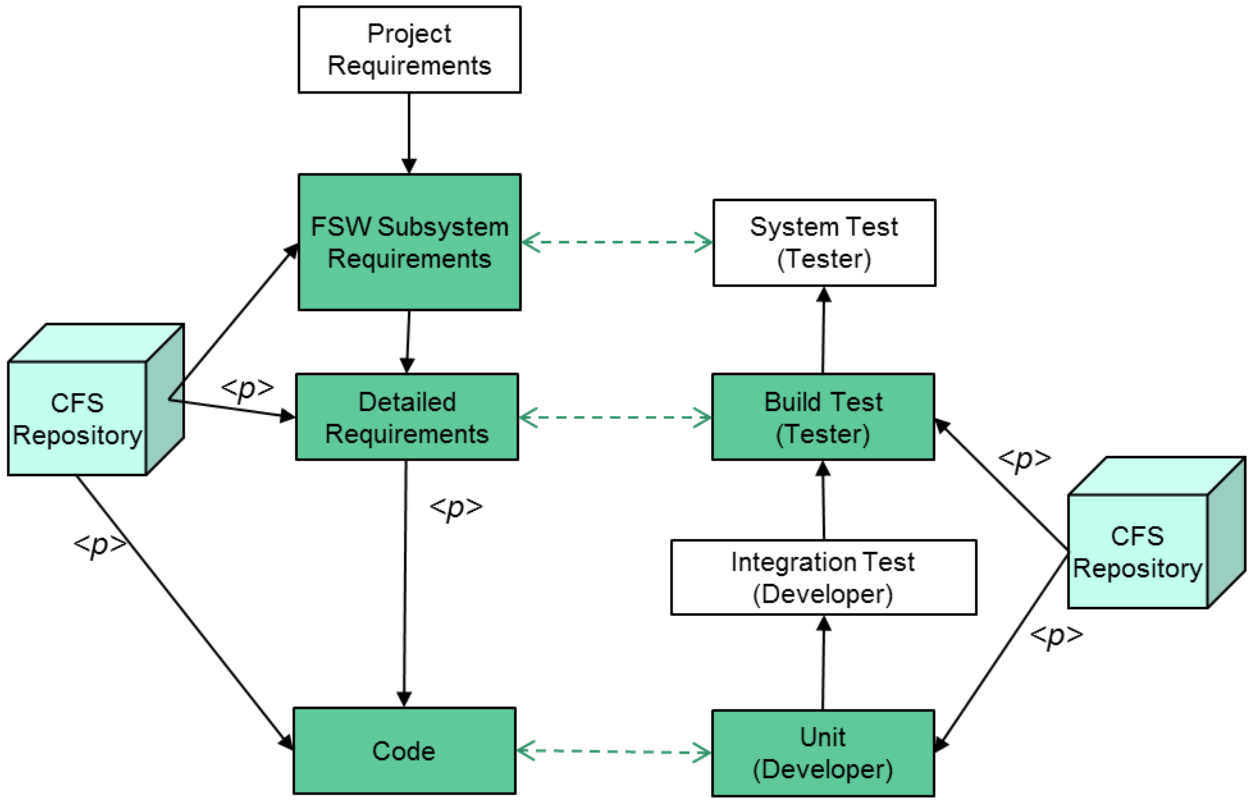


Figure 1-2 – cFS-based Project FSW Lifecycle

All of the artifacts in Figure 1-2's *cFS Repository* boxes are available as part of the open source release. The starter kit's features primarily support the developer's integration activities. There are tools that create a "hello world" application and a corresponding unit test harness. The kit is not intended to be an Integrated Development Environment (IDE) so the user is free to use the development environment of their choosing. Once an application is mature the kit supports the integration of an application into the kit. Build testing support may be added as a future enhancement.

The cFS was originally developed for NASA Software Class B projects as defined by the NASA Procedural Requirement NPR-7150.2B (https://nodis3.gsfc.nasa.gov/main_lib.cfm). These process requirements are defined based on best practices that have produced high quality software. However, the cFS supports a wide range of platforms and therefore a wide range of applications and some applications may not require the same level of rigor. The following steps outline how the artifacts are used in a typical Class B lifecycle.

1. **Requirements Management**
    a. The FSW team receives project requirements. These requirements are traced to existing cFS subsystem requirements. Most if not all of the requirements at this level are implemented by cFS applications so a FSW systems engineer can tailor the cFS to a project by selecting the appropriate cFS applications. These options will continue to grow as the cFS App Store contains more applications.

b. The detailed FSW requirements are *instantiated* by selecting specific configuration parameters for parameterized requirements.

2. Code Instantiation

a. The cFS configuration parameters are contained in C header files that are set by the FSW team. These parameters are refined as the development effort matures.
b. Note in Figure 1-2 that some configuration parameters trace to requirements and some are only contained in the C header files. The header-only parameters are design in nature and do not impact functional requirements. For example default file paths and names are defined as configuration parameters and these do not trace to a functional requirement. These design parameters are verified during system integration.

3. Verification and Validation

a. A project does not typically rerun component unit tests unless a component has been modified. The unit tests have been designed to test all source lines and to provide maximal code path coverage. The current unit tests have not been designed to adapt to project-specific configuration parameters.
b. The current cFS artifacts do not include integration tests. Projects must perform this step to verify the cFS properly functions as a system.
c. The cFS build test verify functional requirements and these have been designed to read in the C header files and adapt the test accordingly so the project-instantiated functional requirements can be verified. However the cFS build tests execute on the Advanced Spacecraft Integration & System Test (ASIST) ground system so if a project is using a different ground system then the build test can't be rerun as delivered.
d. Most GSFC projects perform system level test which are designed based on user scenarios rather than from a functional requirements perspective. The current cFS artifacts do not cover this level of testing.

1.2.1 cFS Architecture

While a majority of the heritage analysis focused on FSW functional features a significant and conscious effort was made to address the cFS's architectural quality attributes such as portability, performance, scalability, interoperability, verifiability, and complexity. Figure 3 illustrates the cFS architecture and two fundamental architectural features are the Application Program Interface (API)-based layers and the definition of an application as a distinct well-defined architectural component. Applications can easily be integrated into the build system and even dynamically added/removed during runtime.

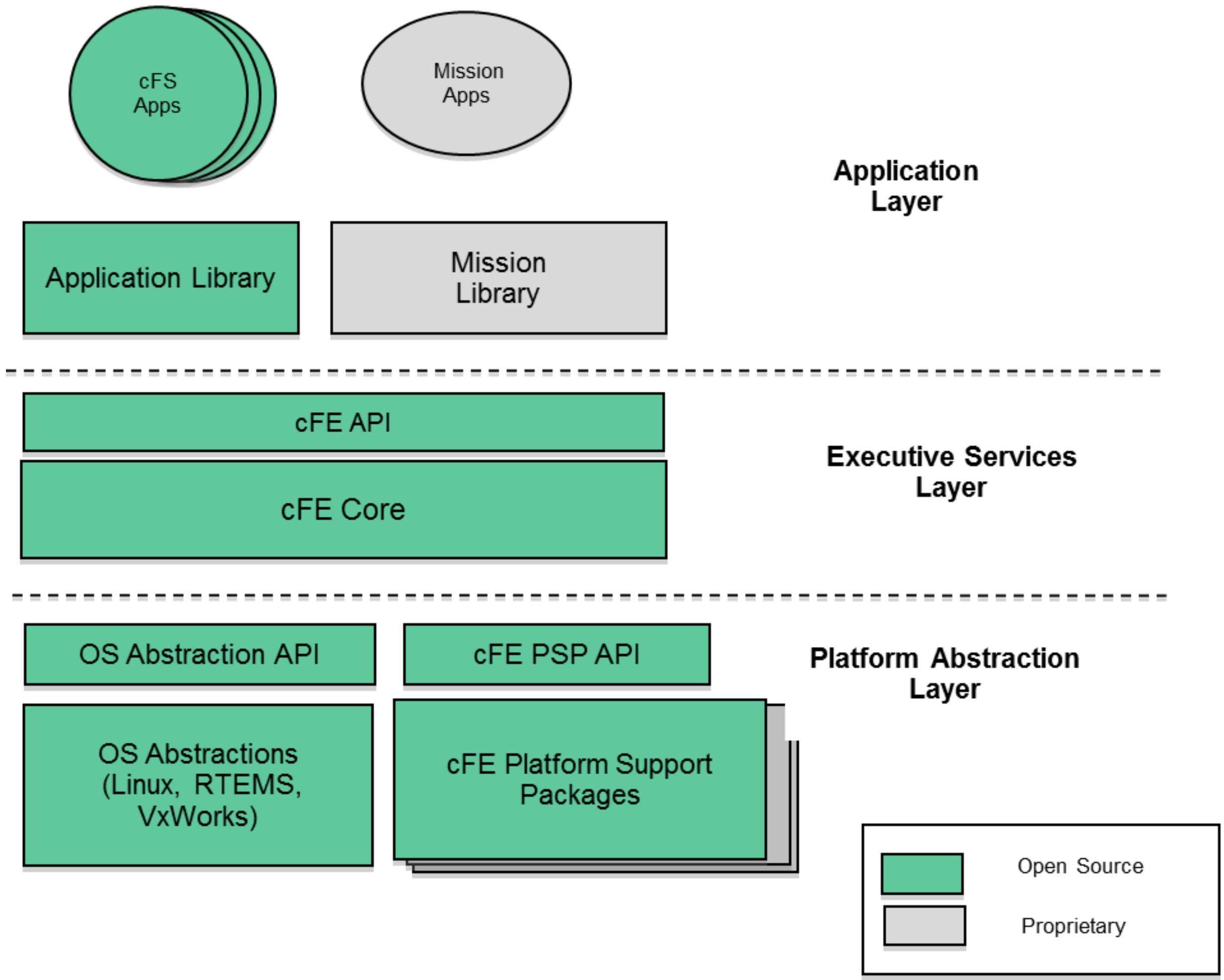


Figure 1-3 – cFS Layered Architecture

The cFS defines 3 layers with an API between each layers. Layer 1 supports portability by decoupling the higher levels from hardware and operating system implementation details. All access to the platform is controlled through two APIs: the Operating System Abstraction Layer (OSAL) and the Platform Support Package (PSP).

Layer 2 contains the core Flight Executive (cFE) that provides five services that were determined to be common across most FSW projects. The core services include a Software Bus (messaging), Time Management, Event Messages (Alerts), Table Management (runtime parameters), and Executive Services (startup and runtime). The Software Bus provides a publish-and-subscribe CCSDS standards-based inter application messaging system that supports single and multi-processor configurations. Time Management provides time services for applications. The Event Message service allows applications to send time-stamped parameterized text messages. Four message classes based on severity are defined and filtering can be applied on a per-class basis. Tables are binary files containing groups of application defined parameters that can be changed during runtime. The table service provides a ground interface for loading and dumping an application's tables. Executive Services provides the runtime environment that allows applications to be managed as an architectural component. All of the services contain tunable compile-time parameters allowing developers to scale the cFE to their needs.

The APIs in Layers 1 and 2 have been instrumental in the cFS' success across multiple platforms and the cFE API has remained unchanged since the launch of the Lunar Reconnaissance Orbiter in 2009. The APIs, their underlying services, and the cFS build tool chain provide the architectural infrastructure that make applications an explicit architectural component. A cFS compliant application will run unchanged regardless of the host platform. The application layer contains thread-based applications as well as libraries (e.g. linear algebra math library) which can be shared among multiple applications.

As shown in Figure 3 all of the source code has been released as open source. The code is managed by a multi-NASA Center configuration control board (CCB) that ensures that the application context will evolved in a controlled manner.

1.2.2 cFS Application Context

The application layer is where the bulk of the cFS scalability and extendibility occurs. Users create new missions using a combination of existing cFS compliant apps (partial or complete reuse) and new mission-specific apps. Just as the cFE provides common FSW services there is a set of apps that provide common higher level functional services. Figure 4 shows the minimal context for a user app on a single processor system. Three 'kit' apps provide the higher level services. The details of why they're kit apps is explained in Section TBD.

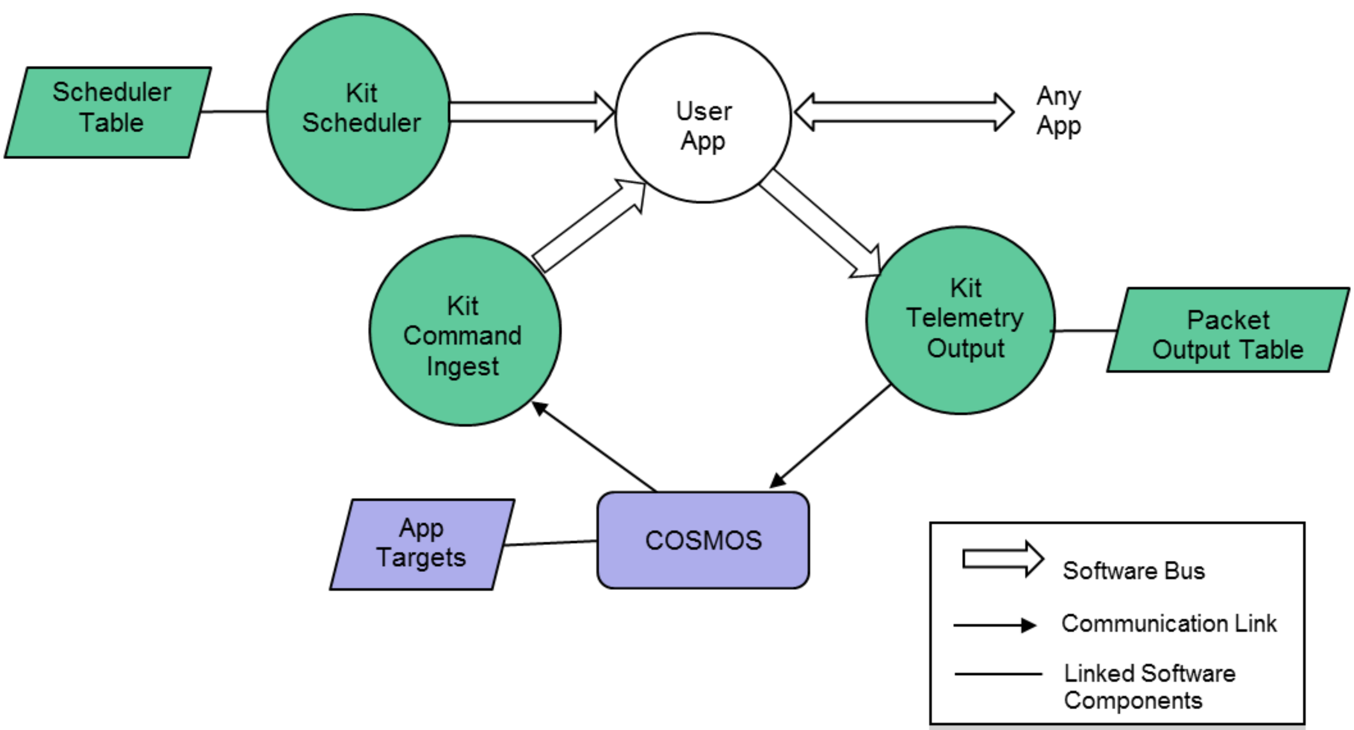


Figure 1-4 – User Application Context

Apps must have the ability to receive commands from and send telemetry to the ground system. The Command Ingest app receives commands from the ground and sends them on the software bus. The software bus uses the command message identifier to route the command to the app that has subscribed to the message id. An app will also generate one or more telemetry packets and send them on the software bus. The Telemetry Output app uses a table to determine which message ids to subscribe to and how often to forward them to the ground system.

Users have multiple mechanisms for how to control the execution of an application. The scheduler app provides a time synchronized mechanism for scheduling application activities. The Scheduler app uses a table to define time slots for when to send a message that users can use to initiate an activity. Activities can be scheduled to occur faster or slower than 1 second. Even if an app's execution is data driven (.i.e. pends for one or more data packets to start its execution) it is often convenient to use the scheduler as a means to send time-based housekeeping telemetry.

1.3 COSMOS Overview

Ball Aerospace's COSMOS is a freely available open source command and control system for operations and test of an embedded system. A set of 15 applications provide automated procedures, real-time and offline telemetry display and graphing, logged data analysis and Comma Separated Variables (CSV) extraction, limits monitoring, command and telemetry handbook creation, and binary file editing. COSMOS scripting offers the full power of the Ruby programming language allowing operators to send commands, verify telemetry, read and write files, access the network, and even send an email upon completion. Advanced debugging functionality allows for single-stepping through procedures, setting breakpoints, and complete logging of all script and user interaction with the system. Detailed data visualization allows for custom screen creation, line and x-y plotting of data, and easy creation of custom 3d visualizations. Offline data analysis and data extraction capabilities make narrowing down anomalies easy.

This user's guide describes the components of COSMOS that are relevant to using the starter kit. For a complete description of COSMOS refer to the documentation at http://cosmosrb.com. The OSK uses the following COSMOS applciations:

- CmdTlmServer
  - Establishes a connection with the system running the cFS. This interface is defined in cosmos/config/tools/cmd_tlm/cmd_tlm_server.txt. Each cFS application is defined as a COSMOS target for the interface.
  - The CmdTlmServer provides a GUI that allows the user to send commands and receive telemetry for each target.
- TlmViewer

    - Organizes and displays custom telemetry pages. The OSK uses this application to display the main OSK page.
- ScriptRunner
    - Provides a ruby script execution environment. The OSK's integration test uses ScriptRunner. This will be replaced by the COSMOS TestRunner application in a future release.

1.4 42 Simulator Overview

42 is an open source software package that simulates spacecraft attitude and orbital dynamics and control. 42 is design to be both powerful and easy to configure and run. It supports multiple spacecraft anywhere in the solar system and each spacecraft is a multi-body model that can be a combination of rigid and flexible bodies. 42 consists of a dynamics engine and a visualization front end. The two components can run on the same processor, different processors, or just the dynamics can be run without visualization.

Figure 5 shows the processing flow of the 42 simulation models. The Ephemeris Models determine object (spacecraft, sun, earth, etc.) positions and velocities in a particular reference frame. This information is input to the Environmental Models that computes the forces and torques exerted on each object. The ephemeris and environmental data is read by the Sensor Models. The FSW algorithms read the sensor data, estimate states, run control laws, and output actuator commands. The Actuator Models compute control forces and torques. The forces and torques from Environmental Models and Actuator Models are input the Dynamics Model that integrates the dynamic equations of motion over a time step. The new states are fed back to the Ephemeris Models and the simulation process is repeated.

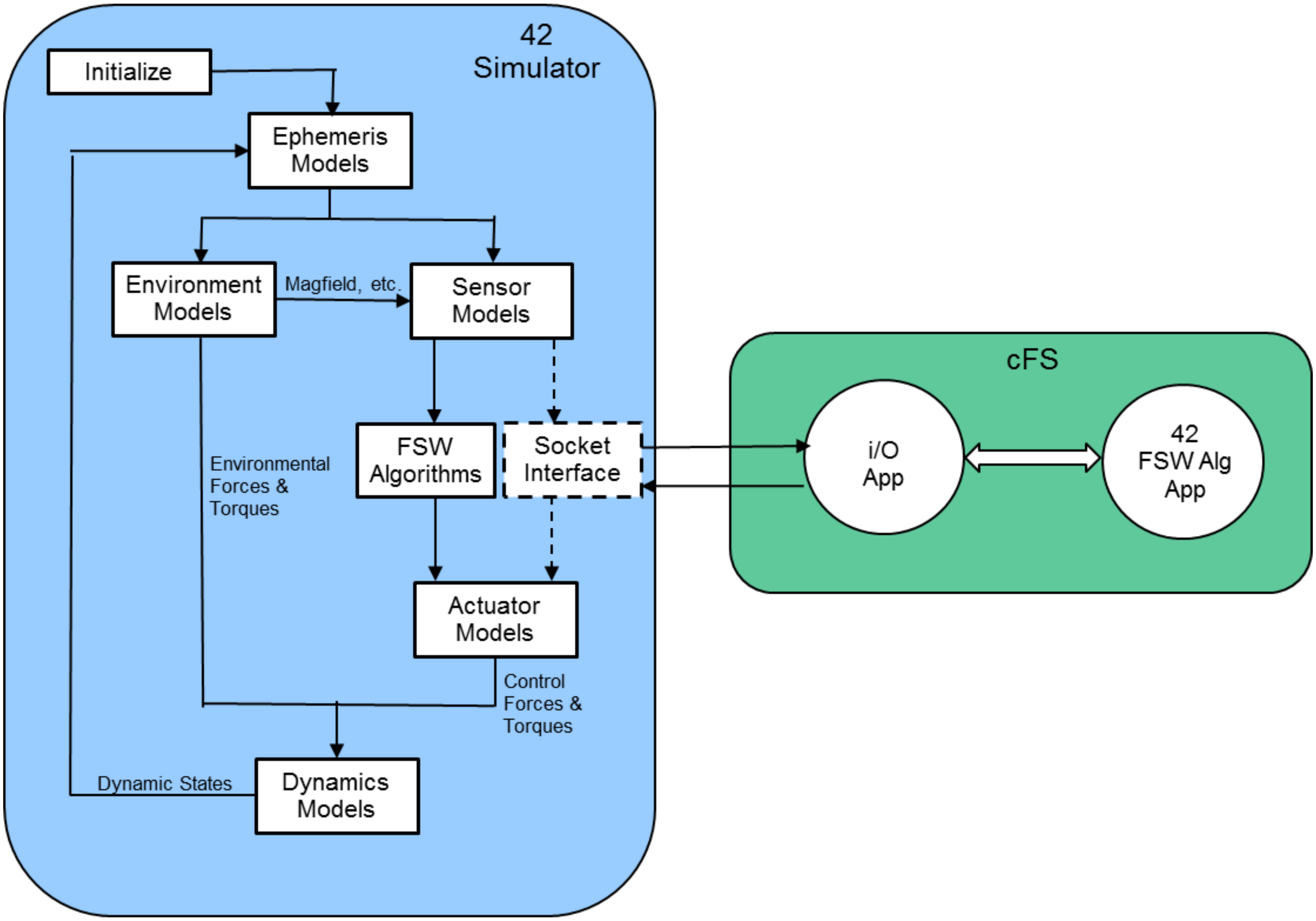


Figure 1-5 – 42 Simulator

The dashed Socket Interface box in Figure 5 has been added to the 42 simulator for the OpenSatKit and replaces the FSW Algorithm box. The FSW Algorithm App running on the cFS implements the 42 FSW algorithms. The I/O App communicates with the new 42 Socket Interface to transfer sensor and actuator data between 42 and the cFS platform. 42 is command line driven which allows it to be controlled by and external program such as COSMOS. This control is not shown in Figure 5.

TODO – Run 42 independently
Zero-length normal in LoadWingsObjFile Aura_MainBody.ojb <--- this printed out many times
TOD - Yes, that's annooying, but harmless. Some 3-D models that I import from outside sources have defects that I had to learn how to live with. Zero-length normals is chief among those.

1.5 PiSat Target

TBD – Describe PiSat

2.0 Using Starter Kit Features

This section describes the starter kit features and how to use them. Everything in the kit is open source so if there's a feature you like you can use the underlying implementation to customize a solution to meet your needs. Note the kit is typically run in a desktop VM and it is not intended to be configured to meet real-time embedded FSW requirements. Therefore it isn't helpful, for example, to create a scheduler table that schedules an app at 10 Hz. The kit assists in functionally integrating apps and Section 5 outlines steps for transitioning from the kit to an embedded system.

2.1 Launching the Starter Kit

After you have installed the starter kit following the instructions in Appendix B you will have the following directory structure as shown in Figure 2-1.

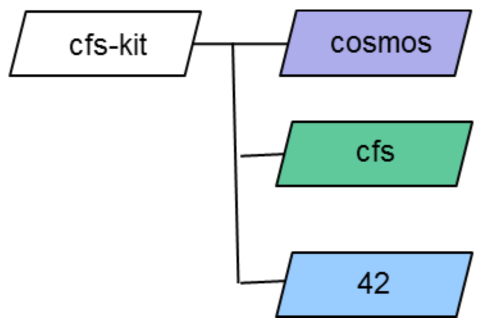


Figure 2-1 – Starter Kit Directory Structure

Start the kit by performing the steps below. When you install the kit for the first time COSMOS is launched as the final installation step so you can start with step 4 in this situation.

1. Create a terminal window by entering
   - ***CTRL-ALT-T***
2. By default you will be in in your home directory. Change the directory to the COSMOS base directory by entering
   - ***cd cfs-kit/cosmos***
3. Start COSMOS
   - Enter ***ruby Launcher***
   - This starts the COSMOS Tool Launcher window as shown on the left side of Figure 2-2.
4. Start the starter kit
   - Click on the ***<cFS Starter Kit>*** button
   - This starts the COSMOS Command and Telemetry Server and Telemetry View tools. The server is needed to connect to the cFS and Telemetry Viewer is sued to launch the kit's the kit's main window as shown on the right side of Figure 2-2.
5. Start the cFS

- Click on the green **_<Start cFS>_** button
- This creates a new terminal window as shown in Figure 2-3 and starts the cFS within in the window. A series of startup messages are displayed. There may be some warnings such as system log buffer full, messages sent with no subscribers, etc. These messages will not affect the functionality of the system and they will be cleaned up in future releases.

6. Enable telemetry by clicking on the **_<Enable Telemetry>_** button. This sends a command to the kit's telemetry output (KIT_TO) app to connect COSMOS. If the command is successful the **_Time_** field will turn white and you'll see time incrementing. If you get a COSMOS connection error then click the **_<Enable Telemetry>_** button as second time.

At this point you have a running system and can start to explore the kit's feature

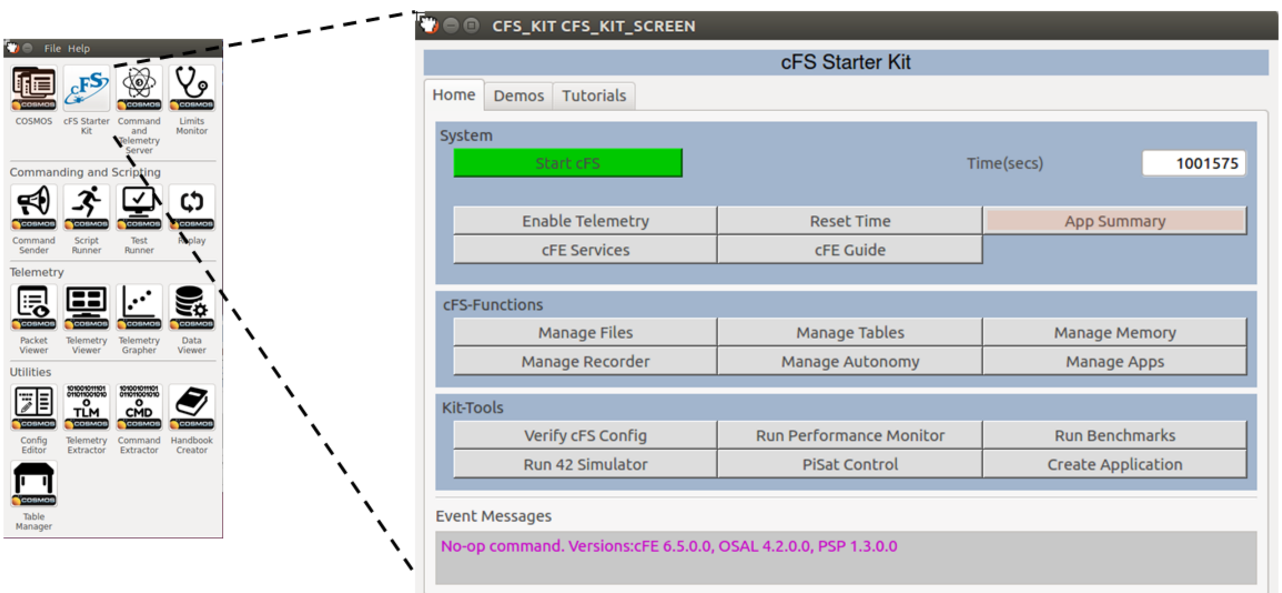


Figure 2-2 – Launcher and Kit's Main Page

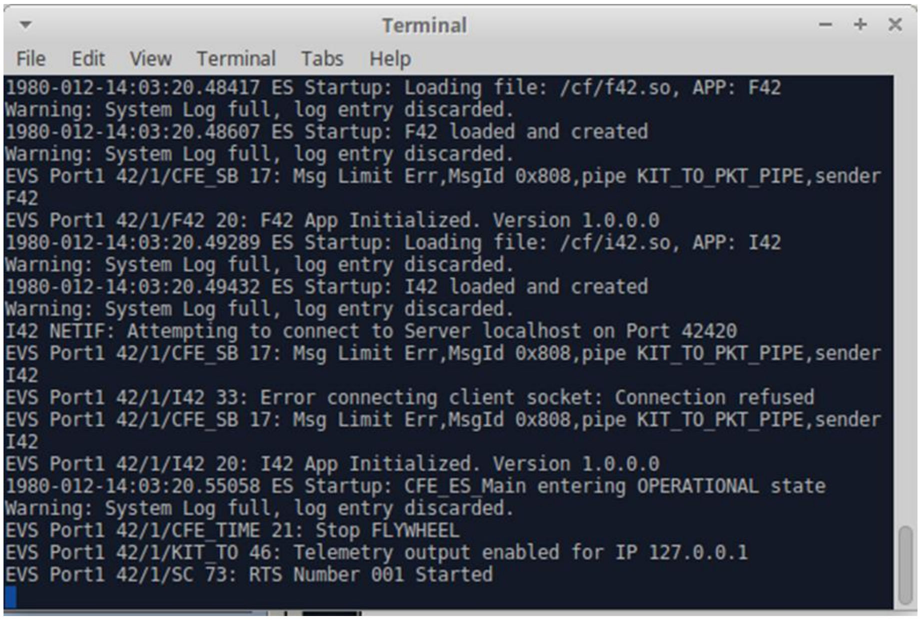


Figure 2-3 – cFS Terminal Window

When the ***<cFS Starter Kit>*** button is clicked two COSMOS services are automatically started: The Command and Telemetry Server and the Telemetry Viewer. Their screens are shown in Figures 2-4 and 2-5, respectively. Both screens can be minimized to reduce clutter but they can't be closed because that would terminate the services. The Command and Telemetry Server manages the communication between the cFS and COSMOS. The Telemetry Viewer manages all of the screens.

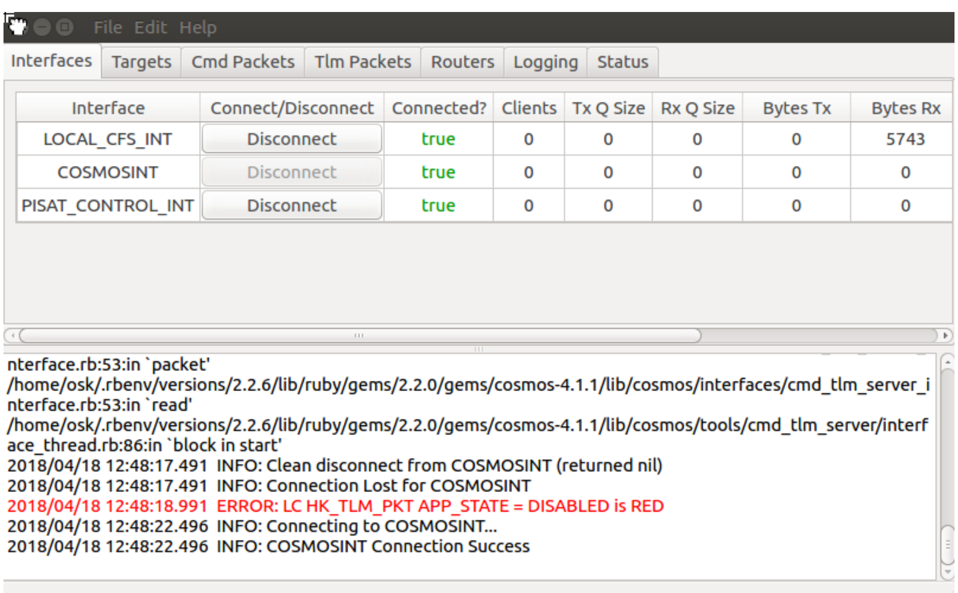


Figure 2-4 – COSMOS Command and Telemetry Server

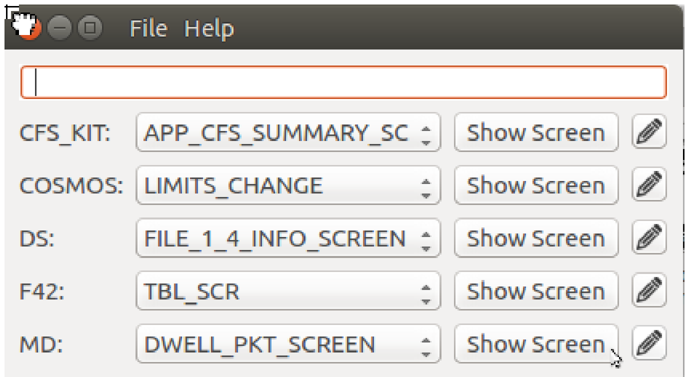


Figure 2-5 – COSMOS Telemetry Viewer

2.2 Kit Feature Overview

The main page layout shown in Figure 2-2 reflects the primary goals of the kit: provide a complete cFS system to simplify the cFS learning curve, simplify the cFS deployment, simplify application development and integration into a cFS system, and assist in porting the cFS to a new platform. The cFS is a complex system so not every cFS feature is covered by the kit. A conscious effort was made to limit the kit's complexity while supporting enough cFS functionality to allow a new user to successfully use the cFS with pre-configured applications. The kit's pages provide access to the most commonly used commands and telemetry. Note that all cFS commands and telemetry are accessible from the COSMOS Command and Telemetry Server tool that is shown in Figure 2-4.

2.3 Kit Main Page

The main page has three tabs: Home, Demo, and Tutorials. The Home tab provides buttons to perform all of the kit's built-in functions. The Demo tab provides pre-configured demonstrations for each of the buttons in the Home tab's cFS-Functions section. The Tutorials tab provides access to script-driven tutorials that can be customized by the user. See Section 5 for more details on Demos and Tutorials.

The Home tab is divided into four sections: System, cFS-Functions, Kit-Tools, and Flight Event Messages. The System section is described in Table 2-1. The cFS Functions and Kit- Tools are described in Sections 2.4 and 2.5 respectively. The Flight Event Message window displays the last event message sent by the FSW.

Button/Field	**Description**
Start cFS	Start the cFS in a terminal window

Time	cFE Executive Service's housekeeping telemetry time seconds value. Should start incrementing after the cFS is started
Enable Telemetry	Telemetry Output's telemetry to COSMOS should be enabled when the cFS is started with the cFS button. If the cFS is running and telemetry has not been enabled this button provides a convenient way to start telemetry.
Reset Time	Resetting time command is a quick and convenient way to show COSMOS and the cFS are communicating properly.
App Summary	Launches two screens. One screen showing all of the cFS apps in the kit and the other screen shows all of the apps defined specifically for OSK. Each app's housekeeping telemetry (see Section 3) sequence counters and command counters ae displayed. This provides a quick look capability to show that all of the apps are running.
cFE Services	Launches a screen that provides a section for each of the cFE services: Executive, Event, Software Bus, Table, and Time. A command button is provided to easily access a service's commands and text fields display the most relevant telemetry. This screen is handy to use in an instructional setting so students can explore the various service features.
cFE Guide	This button launches a web browser displaying the cFE HTML documentation.

Table 1 – Home Page System Buttons and Fields

2.4 cFS Functions

The *cFS Functions* section of the main page contains buttons that launch function-oriented pages. Each page contains buttons to execute the essential commands to perform the functions along with telemetry data to monitor the performance. These functions may be performed by multiple applications and the intention is not to provide every available command and telemetry point. The goal of the pages is to allow the user to explore and learn about how the most common functions can be achieved. The cFS documentation is application oriented so it can be difficult for a user to understand how multiple apps can be used to achieve a desired system function. These pages along with the demos (See Section 5) are intended to help overcome this challenge.

Note in a mission test or operational environment scripts are typically used to ensure repeatable and reliable behavior which reduces the risk of operator error. Each button on these functional pages invokes a ruby script that can be used as a starting point for basic scripting examples but they are not structured for a test or operation scripting framework. Section 5, Implementing a Mission and Section 6, Test Framework, describe how COSMOS's scripting features can be used for testing and operations.

2.4.1 Manage Files

Managing files includes manipulating flight directories and files and transferring files between the ground and the flight file systems. Figure 2-6 shows the file management screen.

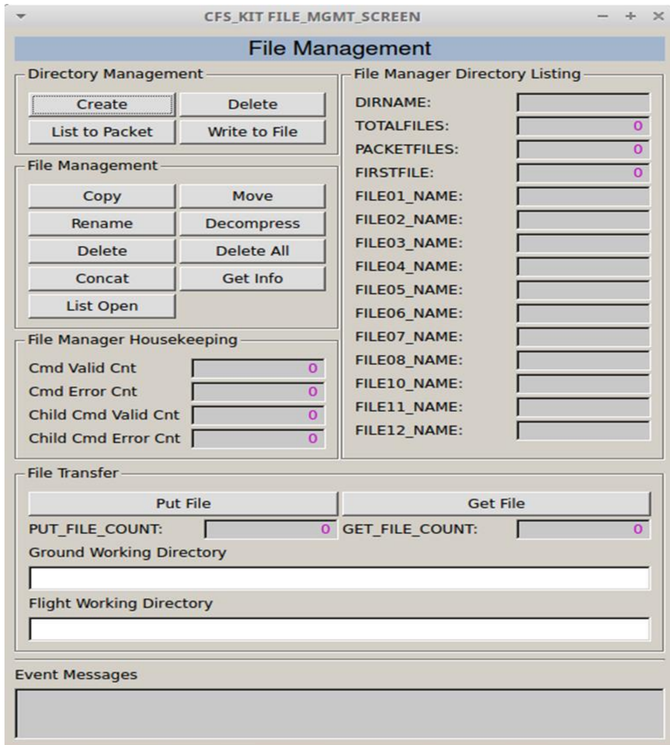


Figure 2-6 – File Management

Directory Management

The default flight working directory is "/cf" which stands for compact flash. In the Linux desktop environment it maps to *~/OpenSatKit/cfs/exe/cpu1/cf*. The default ground working directory is *~/OpenSatKit/cosmos/cfs_kit/file_server*. Note the working directory fields on the screen won't get populated until a transfer takes place.

To list a directory click the **<List to Packet>** button. This invokes File Manger (FM) app's list directory to packet command. The user is prompted for the directory and the starting offset within the directory. The offset is a numerical offset starting at 0 that indexes into an alphabetical directory listing. As you may have guessed the challenge is knowing what offset to use when you don't necessarily know what's in the directory. You can always start with an offset of zero and walk your way through a directory in chunks. The first 12 files in the directory listing are shown on the right side of the screen.

Another option is to write the directory to a file and then use the **<Get File>** button to transfer the file to the ground.

TBD – Display file tool.

The kit maintains a realistic flight-ground separation and requires file transfers even though COSMOS and the cFS reside within the same file system. This makes it easier to use the kit with a remote target.

<u>File Management</u>

2.4.2 Manage Tables

TBD

Figure 2-7 – Table Management

2.4.3 Manage Memory

TBD

Figure 2-8 – Manage Memory

2.4.4 Manage Recorder

TBD

Figure 2-9 – Manage Recorder

2.4.5 Manage Autonomy

TBD

Figure 2-10 – Manage Autonomy

2.4.6 Manage Applications

TBD

Figure 2-11 – Manage Applications

2.5 Kit Tools

2.5.1 Verify cFS Configuration

TBD

2.5.2 Run cFS Performance Monitor

TBD

2.5.3 Run Benchmarks

TBD

2.5.4 Run 42 Simulator

TBD

2.5.5 Add Application

TBD

2.5.6 Manage Hardware Targets

TBD

2.6 Demos

2.7 Pre-installed Applications

There are two categories of pre-installed applications: kit and cFS. The kit apps provide functionality that allows OSK to function as a coherent system and in turn allows OSK to serve as an educational platform. Flight projects would not need all of these apps. The cFS apps provide common flight functionality and would often be included in a flight project. Figure 2-TBD shows all of the OSK apps in a traditional "lollipop" diagram.

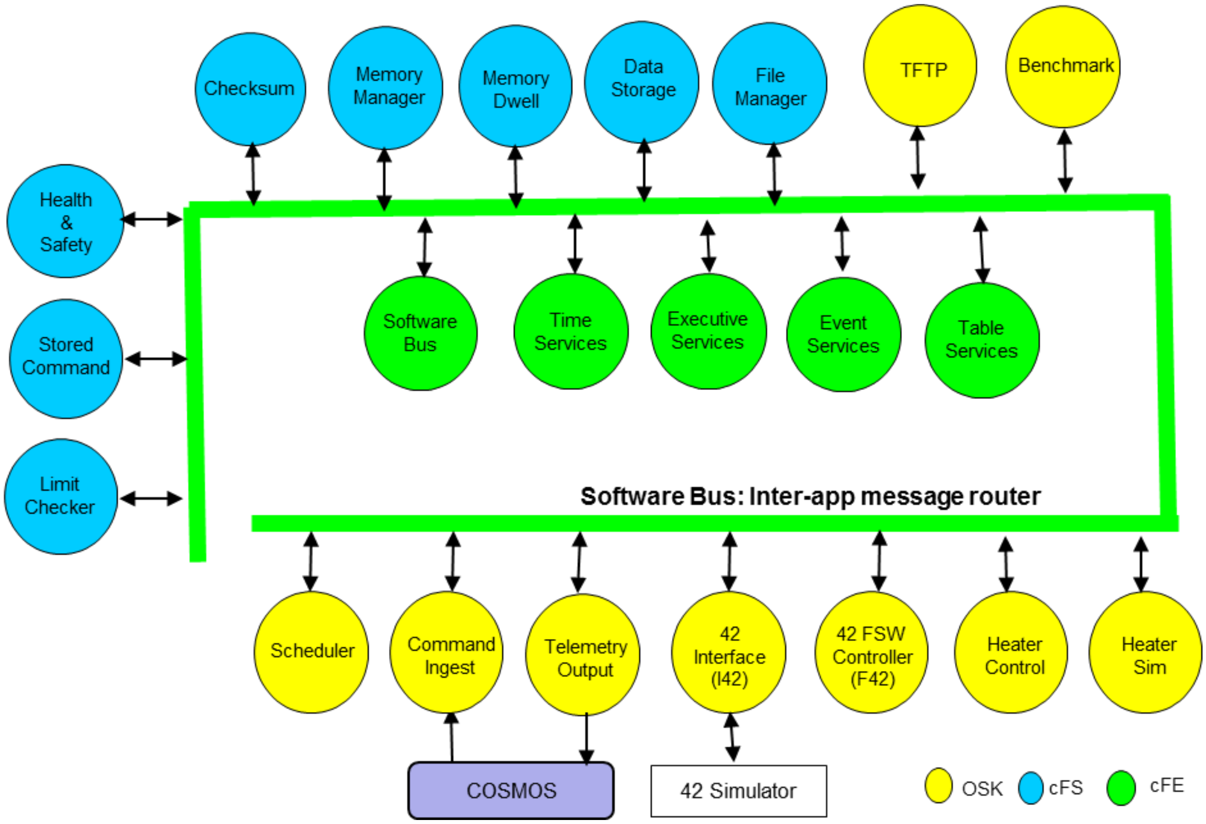


Figure 2-TBD – Kit Apps

2.7.1 Kit Applications

Four of the kit applications provide an operational environment for the system as shown in Figure 2-TBD. Kit Scheduler (KIT_SCH) behaves simialrly to the cFS SCH and SCH_LAB apps. KIT_SCH was primarily created to allow JSON text files to be used fot it's tables. All of the kit apps use JSON files for tables. Refer to the OpenSatKit App Framework guide for details. The default KIT_SCH scheduler table divides a one second period into five 200ms slots and each slot can have up to ten activitiees. An activity identifies which software bus message (defined in KIT_APP's message table) will be sent. OSK is designed as a training tool running on a non-realtime platform so it's schedule table is not very complex. A mission's table could contain orders of magnitude for entries and it's typically driven by the frequency of the attitude determination and control apps.

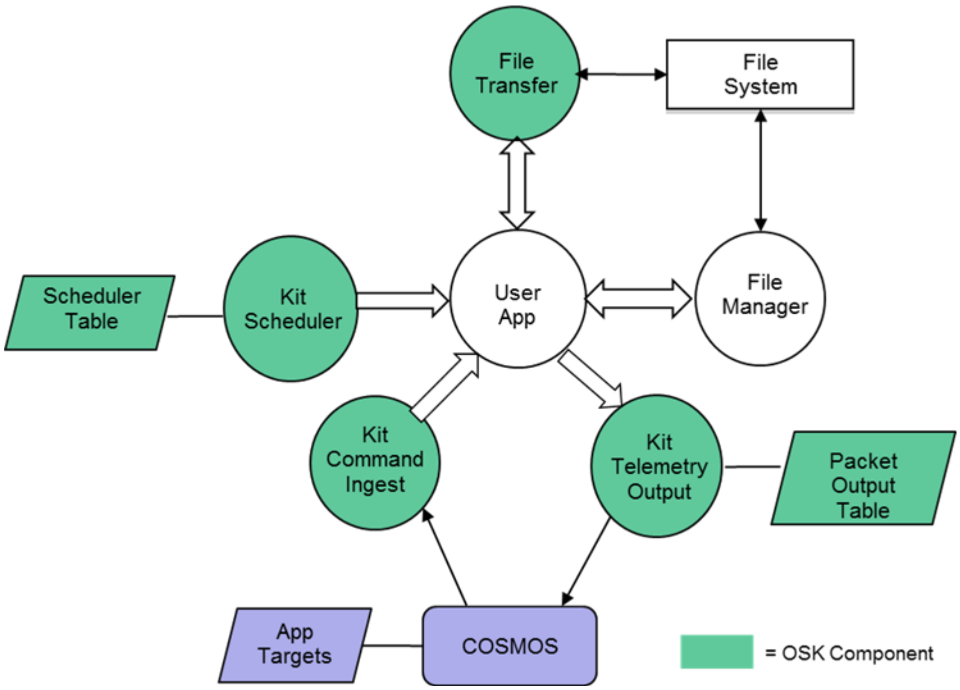


Figure 2-TBD – Kit Operational Apps

Most, if not all apps define a houskeeping (HK) telemetry message containing health and status data. This message is typically sent every 3-5 seconds and the scheduler table is used to schedule the "send housekeeping request" messages that tells an app to send its HK message. Note activities have periods that can be used to schedule activities that occur over multi-second intervals. The decision whether to use KIT_SCH to control an app's execution is driven by an app's purpose. For example, KIT_CI and File Manager (FM) do not use KIT_SCH to determine when to execute because they are both command driven. The Stored Command (SC) app on the other hand needs to execute at a regular frequency to manage the execution of stored commands.

Kit Command Ingest (KIT_CI) and Kit Telmetry Output (KIT_TO) are used to receive commands from and sent telemetry to COSMOS, resepctively. They both use UDP sockets for communication. When KIT_CI is loaded it automatically connects to COSMOS. KIT_TO does not not automatically establish a connection. A KIT_TO Enable Telemetry command containing the COSMOS IP address and port number must be sent to start telemetry. This is automatically done when the cFS is launched for OSK's main screen using the "Start cFS" button. KIT_TO's packet output table defines which telemetry packets are output by KIT_TO. KIT_TO provides a commands that allows a telemetry packet already defined in its tabel to be enabled or disabled.

The Trivial File Transfer Protocol (TFTP) app is the fourth kit app that completes the operational environment. TFTP is used to send files to and receive files from COSMOS, respectively. TFTP was chosen over the open source CFDP app CF because a COSMOS compatible CFDP engine did not exist at the time OSK was created. TFTP uses UDP sockets for it's communication as opposed to the command and telemtry packet interface that CF uses. This is not seen as a limitation for OSK because it is only inetdned to be used in a training environment.

The Interface-to-42 (I42) and 42 FSW (F42) apps work together to

The Heater Control (HC) and Heater Simulation (HSIM) apps are included for demonstration purposes. TBD – Describe HC and HSIM data flow.

TBD – Describe benchmark app after it gets redesigned.

2.7.2 cFS Applications

Eight cFS apps are included in OSK and they were chosen because they are highly likely to be used on a mission. This secton only provides a brief overview. Refer to each application's user's guide for a complete description. The fulfill three functional categories: data mangement, autonomy, and health & safety.

Data Mangement

File Manager (FM) and Data Storage (DS) provide data management services. FM provides a ground interface for configuring directories and files. See section 2.4.1 Manage Files for details. DS provides a service for recording packets into files. Its table defines which packets are stored in which files, how the files are named, and criteria for when a file is closed and a new one is opened.

TBD – Create and describe a default DS configuration.

Autonomy

The Limit Checker (LC) and Stored Command (SC) apps are used to perform autonomous operations. See Section 2.4.5 Manage Autonomy for details.

TDB – Describe default configuration

Health & Safety

Checksum
Memory Dwell
Memory Manager
Health & Safety

3.0 Demos and Tutorials

Demos and tutorials are included to improve the educational aspects of the kit. OSK contains a fixed set of preconfigured demos. The tutorial framework design allows users to customize OSK with their own material. Figure 5-1 highlights the relevant directories. The Demos and Tutorials subsections provide details on each feature.

```
cosmos
|- config
|   |- targets
|   |   ...
|   |   |- CFS_KIT
|   |       |- lib
|   |       |- screens
|   |   ...
|
|- cfs_kit
|   |- tutorials
|       |- osk_tutorial.json
|       |- tutorial_x
|           |- lesson_a.rb
|           |- lesson_b.rb
|
|- lib
    |   ...
```

Figure 5-1 – Demo & Tutorial Directories

3.1 Demos

Demos are screen-driven scripts that are intended to be self-guided. They can be launched from the Demos tab as shown in Figure 5-2. Figure 5-3 shows the initial screen when the File Management Demo is launched. The demo screens are defined CFS_KIT/screens target directory and the Ruby scripts that control the demos are defined in the CFS_KIT/lib directory. Demos are pre-canned scripts that are not intended to be modified by the user but since this is an open source project nothing precludes the user from modifying or extending them.

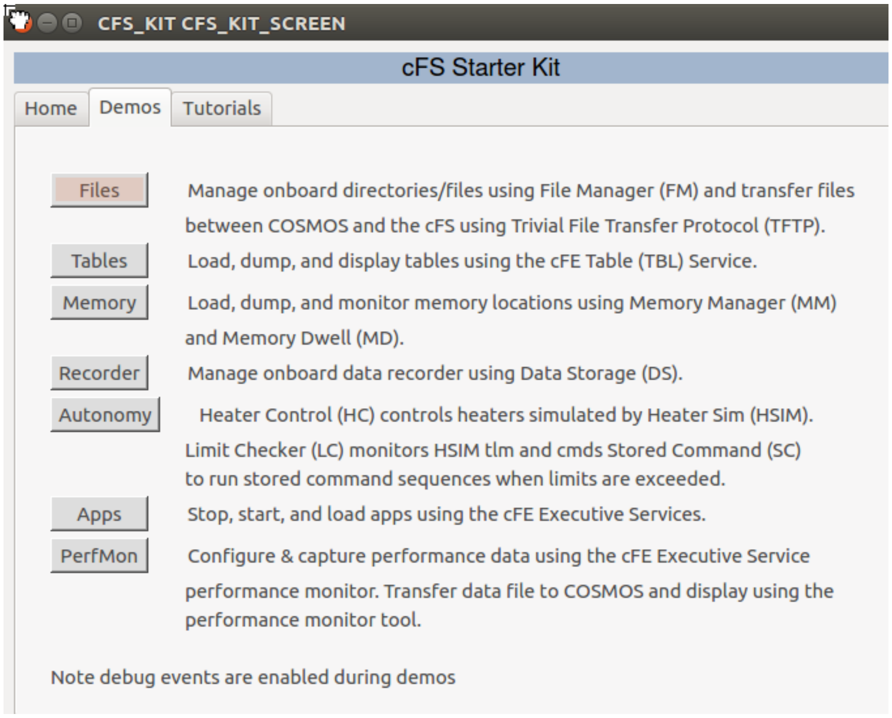


Figure 5-2 – Demo Main Screen

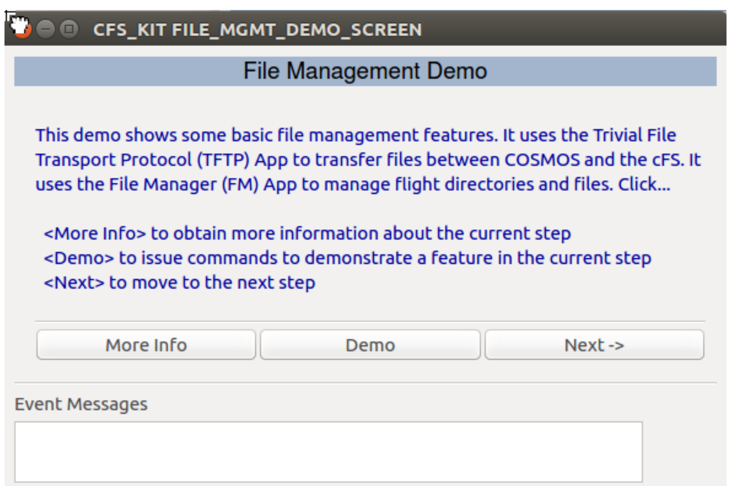
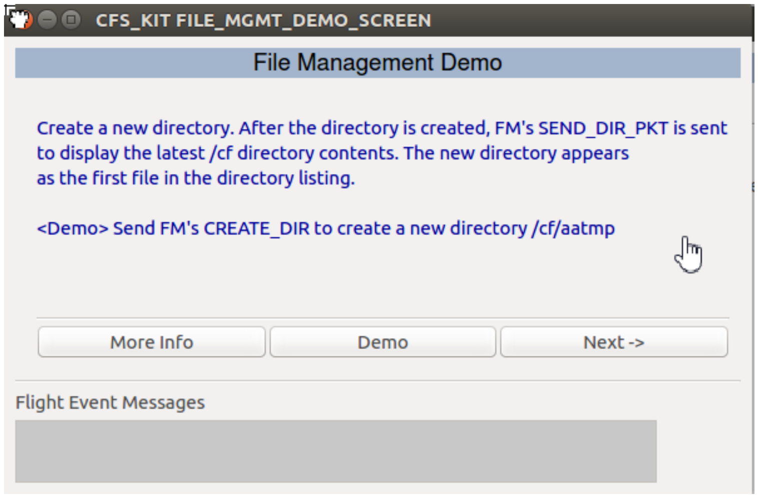
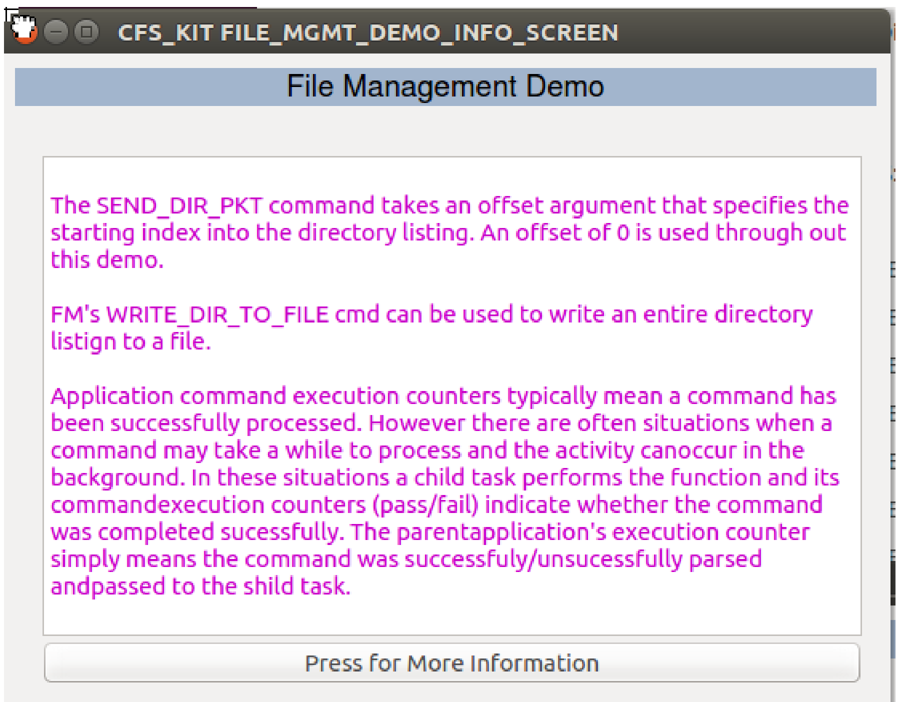


Figure 5-3 – File Management Initial Demo Screen





3.2 Tutorials

The tutorial framework allows users to extend OSK with their own material making it very easy to tailor OSK to serve a particular educational purpose. User material formats include HTML, PDF, and Ruby scripts. The file *cfs_kit/tutorials/osk_tutorials.json* shown in Figure 5-4, defines the tutorials. Each entry in the "tutorials" array defines a tutorial. A tutorial has one or more lessons. The lessons defined in the "lessons" array are displayed in a drop down menu. Each tutorial is defined in its own directory and each lesson is defined its own file. Tutorial formats include "SCRIPT", "HTML", and "PDF". All of the lessons in a tutorial must be in the same format.

```
{
  "tutorials": [
  {
    "name":   "cFE",
    "directory": "cfe",
    "button": "cFE",
    "description": "Core Flight Executive",
    "user-prompt": "Select tutorial",
    "format": "SCRIPT"
    "lessons": ["ES","TBL"]
  },
  {
    "name":   "Operational Applications",
    "directory": "op_apps",
    "button": "Op Apps",
    "description": "Apps that provide an operational runtime environment",
    "user-prompt": "Select tutorial",
    "format": "SCRIPT"
    "lessons": ["KIT_CI","KIT_SCH"," KIT_TO"]
  },
  {
    "name":   "Developing Apps",
    "directory": "dev-apps",
    "button": "Dev Apps",
    "description": "Exercises to demonstrate how to develop an app",
    "user-prompt": "Select tutorial",
    "format": "HTML"
    "lessons": ["INTRO"]
  }
  ]
}
```

Figure 5-4 –Tutorial Definition File osk_tutorials.json

Perform the following steps to run a tutorial:

1. Navigate to the Tutorials Tab screen shown in Figure 5-5 and select <Tutorials> that will launch the tutorial main screen shown in Figure 5-6. This menu may not look exactly like your menu if tutorials have been added or removed. As you can see the JSON "button" definition is used to label the button and the "description" definition is used in the text to the right of the button.
2. When you select a tutorial button, the tutorial's lesson message box will appear that provides a drop down menu with each of the lessons as shown in Figure 5-7.
3. Select a lesson to the launch the appropriate application (based on the format) to start the lesson.
    a. The lesson file is located in cosmos/cfs_kit/tutorials/"directory"/"lesson".xx.
    b. In the cFE example if the user selected table services (TBL) then the cosmos/cfs_kit/tutorials/cfe/tbl.rb is launched in the COSMOS Script Runner.

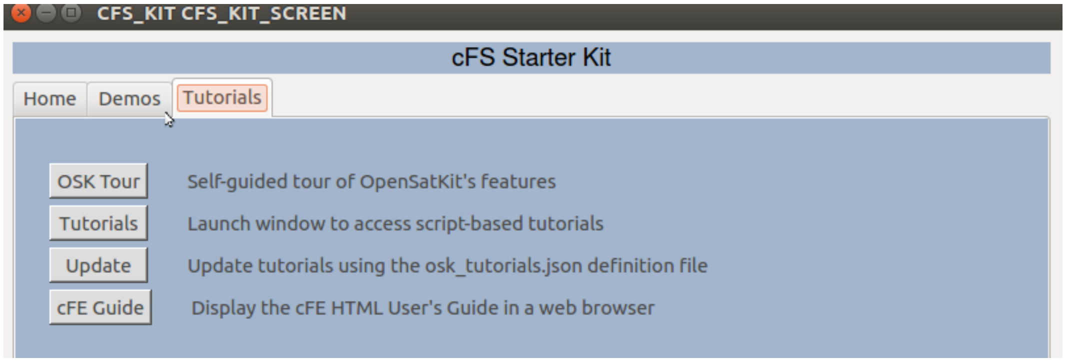


Figure 5-5 – Tutorial Tab Screen

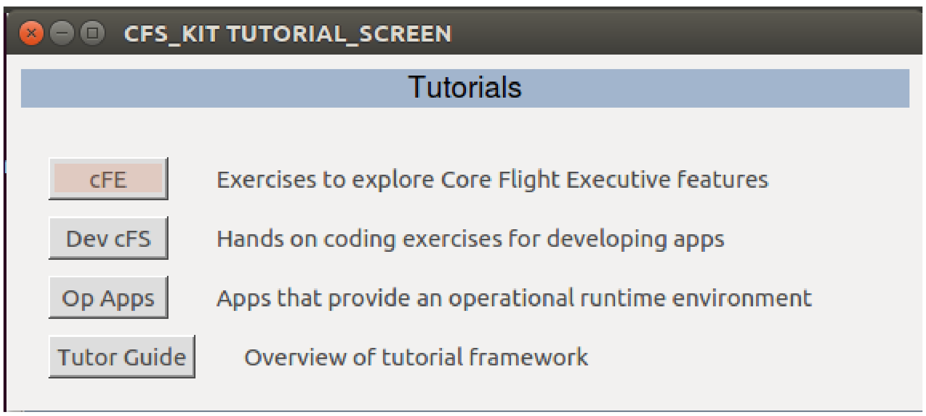


Figure 5-6 – Tutorial Main Screen

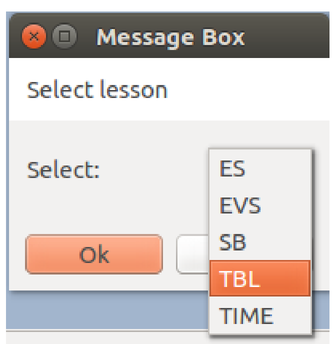


Figure 5-7 – cFE Tutorial Lesson Selection Message Box

Perform the following steps to create a new tutorial:

- Create a new directory in the cfs_kit/tutorials directory
- In your new directory create one file for each lesson

- Create a new tutorial entry in the "tutorials" array in *osk_tutorials.json*
- Navigate to the Tutorial Tab Screen shown in Figure 5-5 and select <Update>
    a. This creates a new turorial_screen.txt file in the *CFS_KIT/screens* target directory. The previous turorial_screen.txt file is preserved in *tutorial_screen_year_month_day_hourminutesecond.txt* file
- If a tutorial is added without any lesson files defined, the screen will be created but the user will get an error if they try to launch a lesson in a tutorial without a corresponding file.

Remove tutorials by removing its definition from the JSON file. Permanently remove them by deleting the corresponding directory and lesson files.

4.0 Managing Applications

This section describes how to create, compile/link, add, and remove applications from within the kit. All of the steps needed to perform these functions are described, however, it is not intended to be a comphrensive application developer's guide. The *cFE Application Developers Guide.doc* located in *cfs/cfe/docs* provides a complete description of developing apps for the cFS.

4.1 cFS Application Build Environment

The cfs directories that you'll need to work with for managing apps and building the system are show in Figure 3-1. The *cfs/apps* directory contains the source code for each app. The cFS build system does not require the apps to be physically located in the same base directory but the kit is structured this way for convenience. The *apps* subdirectories are described in the *cFE Application Developers Guide.* The *for_build* directory is not required, because the kit uses the cmake build environment and the *for_build* directory is used by the cFS "classic build" system which will eventually be deprecated. The *tables* subdirectories is only required if the app has a table. Note the OpenSatKit apps follow a different design pattern than the cFS apps. An overview of the kit app design is provided in Section 5 and the *OpenSatKit Application Developer's Guide* contains a complete description.

```
cfs
|- apps
|   |- aaa
|      |- for_build
|      |- mission_inc
|      |- platform_inc
|      |- src
|      |- table
|      |- unit_test
|   |- bbb
|   ...
|- build
|   |- cpu1/...
|   |- exe/cpu1/cf
|
|- osk_def (was sample_def)
```

Figure 3-1 – cFS Application Directories

The *cfs/build/cpu1* directory is automatically generated by the cmake preparation step. When OpenSatKit is installed this preparation step is automatically performed. The *cfs/build/exe/cpu1/cf* directory is where the build system locates the cFS binary image, application object files, and table files. The *cf* directory stands for compact flash because it is the directory used to boot the cFS. The cmake build system supports building for multiple targets. The kit is configured to build the default configuration for a single cpu named *cpu1*.

4.1.1 Building and installing the cFS

When you first install OpenSatKit the cFS is automatically built for you. If you modify any existing source files you can build the cFS by performing the following steps:

1. Change your directory to *OpenSatKit/cfs*
2. *OpenSatKit/cfs$* **make**
   a. This rebuilds the files but does not install them into *cfs/build/exe/cpu1/cf*
3. *OpenSatKit/cfs$* **make install**
   a. This rebuilds the files and installs them into *cfs/build/exe/cpu1/cf*
   b. The installation process copies files in the *OpenSatKit/cfs/osk_def* directory prefixed with "cpu1_" into *cfs/build/exe/cpu1/cf.* This means if you changed cfe_es_startup.scr or any table files in the /cf directory they will be overwritten.

Table 3-1 identifies key directories and files used in the building and installation of the cFS.

Directory/File	Purpose
/cfs/osk_defs	Directory that contains key files used by cmake to build and install the cFS
targets.cmake	Defines the cFS targets (with build rules) and apps to be built for the target. Also lists files to be copied from osk_defs to the target 'cf' directory. The kit uses target 1 (TGT1) and the important definitions are SET(TGT1_NAME cpu1) – Identifies the targets name SET(TGT1_APPLIST …) – Apps to be built and installed SET(TGT1_FILELIST …) – Files prefixed with "cpu1_" to be copied to 'cf'
cpu1_*.*	Identifies files for the cpu1 target. If they are listed in targets.cmake's TGT1_FILELIST they will be copied to the 'cf' directory
cpu1_cfe_es_startup.scr	This startup script must be defined. It defines the apps that will be loaded by the cFE during initialization.
cfs/apps/xxx/ cmakelist.txt	Each app is contained within its own apps directory and at the cmakelist.txt must be present in the top-level app directory. This file defines what gets compiled for the app and any dependencies. See the example in Figure 3-2.
cfs/build/exe/cpu1/cf	"Compact Flash" directory that contains all of the files necessary to run the cFS.

Table 3-1 – cFS Application Build Directories

Figure 3-2 shows an example *cmakelist.txt* file that must be included in the top-level directory for each app. The `include_directories` statements define the search paths used by cmake to locate include files. The `aux_source_directory` statements define directory paths that contain the source code to build the app and populate the symbol passed to the function. The `add_cfe_app` statement adds the specified app to the cmake build. The apps object file name is the same name as the first function parameter. The `add_cfe_tables` statement only needs to be present if the app has cFE style tables and it causes the app's tables to be built.

```
cmake_minimum_required(VERSION 2.6.4)
project(CFS_HS C)

include_directories(fsw/src)
include_directories(fsw/mission_inc)
include_directories(fsw/platform_inc)

aux_source_directory(fsw/src APP_SRC_FILES)
aux_source_directory(fsw/tables APP_TABLE_FILES)

# Create the app module
add_cfe_app(hs ${APP_SRC_FILES})
add_cfe_tables(hs ${APP_TABLE_FILES})
```

Figure 3-2 – Example cFS App cmakelist.txt

Figure 3-3 shows a *cmakelist.txt* file for an app using the OSK app design pattern. The main differences are the cFS's library and OSK's app framework include paths must be included and the `add_cfe_tables` statement is not needed since the app manages its own table files. See *OpenSatKit Application Developer's Guide* for more details.

```
cmake_minimum_required(VERSION 2.6.4)
project(CFS_OSK_DEMO C)

include_directories(fsw/src)
include_directories(fsw/mission_inc)
include_directories(fsw/platform_inc)
include_directories(${cfs_lib_MISSION_DIR}/fsw/public_inc)
include_directories(${app_fw_lib_MISSION_DIR}/fsw/platform_inc)
include_directories(${app_fw_lib_MISSION_DIR}/fsw/mission_inc)

aux_source_directory(fsw/src APP_SRC_FILES)
aux_source_directory(fsw/tables APP_TABLE_FILES)

# Create the app module
add_cfe_app(osk_demo ${APP_SRC_FILES})
```

Figure 3-3 – Example OSK App cmakelist.txt

Note if the application requires the math libraries to be linked in then after the add_cfe_app() line include the following function call: target_link_libraries(my_app m), where "my_app" is the same app name used in the add_cfe_app() call.

4.2 COSMOS Application Definition Environment

The COSMOS directory structure shown in Figure 3-4 highlights the most relevant OSK COSMOS directories for installing applications. See Section 5 for a more complete COSMOS description.

```
cosmos
|- config
|   |- targets
|   |   ...
|   |   |- APP
|   |      |- cmd_tlm
|   |      |- lib
|   |      |- screens
|   |   ...
|   |- tools
|      |-cmd_tlm_server.txt
|
|- lib
   |- cfs_kit_config.rb
   |- message_ids.rb
```

Figure 3-4 – COSMOS Directories

Each cFS app is defined as a COSMOS target in the *cosmos/config/targets* directory. The APP directory must be uppercase and within the APP directory is a cmd_tlm directory that contains files that define the app's command and telemetry packets. . Figures 3-5 and 3-6 show part of the File Manager app's command and telemetry definitions, respectively. The OSK uses Embedded RuBy (ERB), delineated by "<%" and "%>", to standardize and simplify the packet definitions. *cfs_kit_config.rb* is contained in the *cosmos/lib* directory and serves as the single point for all OSK configuration definitions/settings. Any dependencies are also contained with the *cosmos/lib* directory. *message_ids.rb* defines all message IDs and the convention is to have "MID" in the symbol name. In the two examples, FM_CMD_MID and FM_HK_TLM_MID, are defined in *message_Ids.rb*. The cFS is evolving towards adopting the CCSDS Electronic Data Sheets (EDS) standard and it is anticipated that the *message_ids.rb* file will be automatically generated.

```
<%
  require 'cfs_kit_config'

  @APP_PREFIX_STR = "FM"
  @CMD_MID_STR    = "FM_CMD_MID"
  @DEF_FILENAME   = "default"
  @DEF_DIR_NAME   = "/cf"
%>

COMMAND FM NOOP <%= CfsKitConfig.processor_endian %> "Comment"
  <%= CfsKitConfig.cmd_hdr(@APP_PREFIX_STR, @CMD_MID_STR, 0, 0) %>

COMMAND FM RESET_CTRS <%= CfsKitConfig.processor_endian %> "Comment"
  <%= CfsKitConfig.cmd_hdr(@APP_PREFIX_STR, @CMD_MID_STR, 1, 0) %>

COMMAND FM COPY_FILE <%= CfsKitConfig.processor_endian %> "Comment"
  <%= CfsKitConfig.cmd_hdr(@APP_PREFIX_STR, @CMD_MID_STR, 2, 130) %>
  APPEND_PARAMETER OVERWRITE 16 UINT MIN_UINT16 MAX_UINT16 0 "Allow overwrite"
  APPEND_PARAMETER SOURCE   512 STRING <%= @DEF_FILENAME %>  "Source filename"
  APPEND_PARAMETER TARGET   512 STRING <%= @DEF_FILENAME %>  "Target filename"
```

Figure 3-5 – Example Command Definition

```
<%
  require 'cfs_kit_config'

  @APP_PREFIX_STR         = "FM"
  @HK_TLM_MID_STR         = "FM_HK_TLM_MID"
  @FILE_INFO_TLM_MID_STR  = "FM_FILE_INFO_TLM_MID"
  @DIR_LIST_TLM_MID_STR   = "FM_DIR_LIST_TLM_MID"
  @OPEN_FILES_TLM_MID_STR = "FM_OPEN_FILES_TLM_MID"
  @FREE_SPACE_TLM_MID_STR = "FM_FREE_SPACE_TLM_MID"

%>

TELEMETRY FM HK_TLM_PKT <%= CfsKitConfig.processor_endian %> "Comment”
  <%= CfsKitConfig.tlm_hdr(@APP_PREFIX_STR, @HK_TLM_MID_STR) %>
  APPEND_ITEM CMD_VALID_COUNT 8 UINT "Application command counter"
  APPEND_ITEM CMD_ERROR_COUNT 8 UINT "Application command error counter"
  APPEND_ITEM SPARE           8 UINT "Unused”
  APPEND_ITEM NUMOPENFILES    8 UINT "Number of open files in the system"
  APPEND_ITEM CHILDCMDCOUNTER 8 UINT "Child task command counter"
  APPEND_ITEM CHILDCMDERRCOUNTER  8 UINT "Child task command error counter"
  APPEND_ITEM CHILDCMDWARNCOUNTER 8 UINT "Child task command warning counter"
  APPEND_ITEM CHILDQUEUECOUNT 8 UINT "Number of pending commands in queue"
  APPEND_ITEM CHILDCURRENTCC  8 UINT "Command code currently executing"
  APPEND_ITEM CHILDPREVIOUSCC 8 UINT "Command code previously executed"
```

Figure 3-6 – Example Telemetry Definition

4.3 OpenSatKit Application Runtime Environment

Figure 3-7 shows the runtime context for a user application. It's important to understand this context because it identifies all of a user application's interfaces that must be resolved in order for it to operate properly. Note that all of these interfaces are to other applications which means an organization can standardize on a different set of applications thus altering the context.

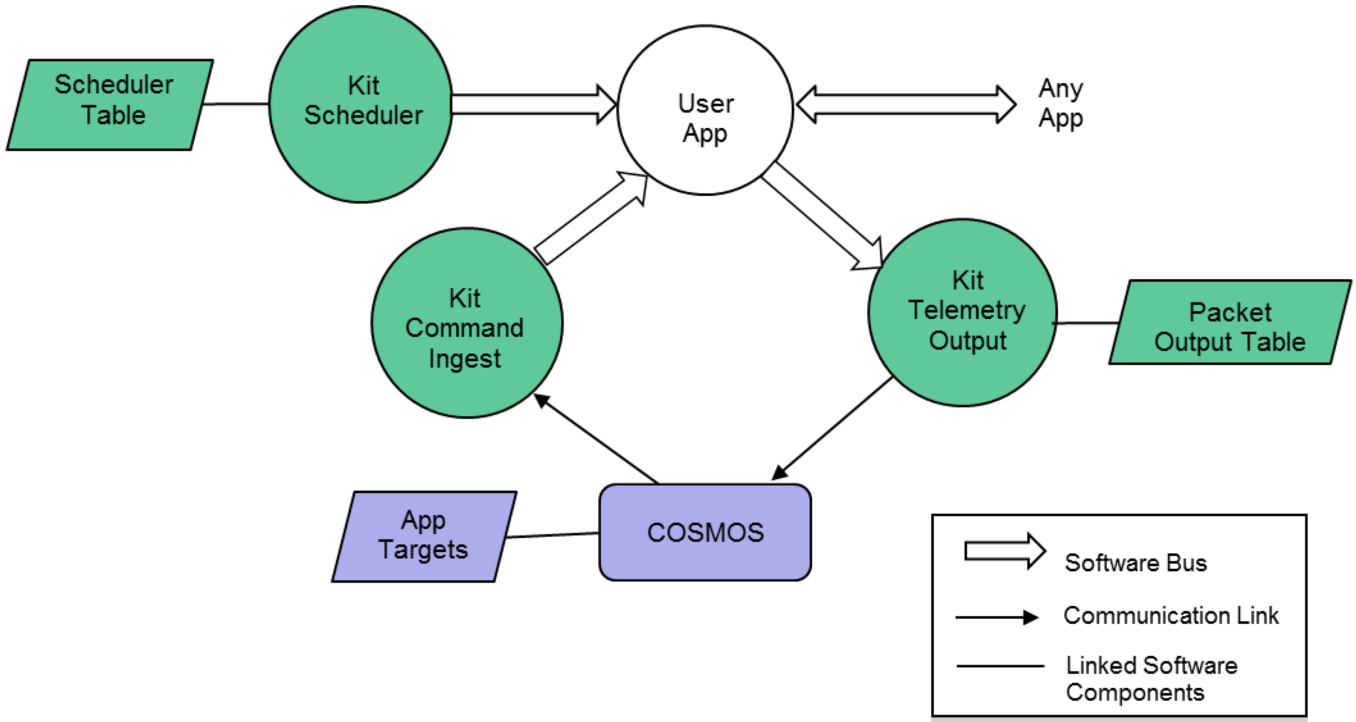


Figure 3-7 – User Application Context

Apps must have the ability to receive commands from and send telemetry to the ground system. The Kit Command Ingest app receives commands from the ground and sends them on the software bus. The software bus uses the command message identifier to route the command to the app that has subscribed to the message id. An app also generates one or more telemetry packets and sends them on the software bus. The Kit Telemetry Output app uses a table to determine which message ids to subscribe to and how often to forward them to the ground system.

Users have multiple mechanisms for how to control the execution of an application. The Kit Scheduler app provides a time synchronized mechanism for scheduling application activities. The Kit Scheduler app uses a table to define time slots for when to send a message that users can use to initiate an activity. Activities can be scheduled to occur faster or slower than 1 second. Even if an app's execution is data driven (.i.e. pends for one or more data packets to start its execution) it is often convenient to use the scheduler as control mechanism for when to send time-based "housekeeping" telemetry.

The kit apps in Figure 3-7 perform the same functions as the "lab" apps released with the cFE. However the kit apps use text files for tables which simplifies the automation of integrating an app into the kit. The kit scheduler has not been qualified for flight so a user will have to transition from the kit SCH to the cFS SCH app. The cFE only provides CI_LAB and TO_LAB which are not flight qualified so the kit apps do not create additional work since every user must develop their own CI and TO apps.

4.4 Creating New Applications

The easiest way to create and integrate a new app is to use the *Create Application* tool. This tool is launched from the kit's main page's home tab using the *Create Application* button in the Kit Tools section. Figure 3-8 shows the *Create Application* screen. Follow the instructions in the main screen's text box. These instructions assume you named your app "*example*".

- Note when naming your app you must avoid cmake keywords such as "test".
- The "Install App" feature has not ben implemented. Follw the instructions after Figure 3-9 to install the new application.

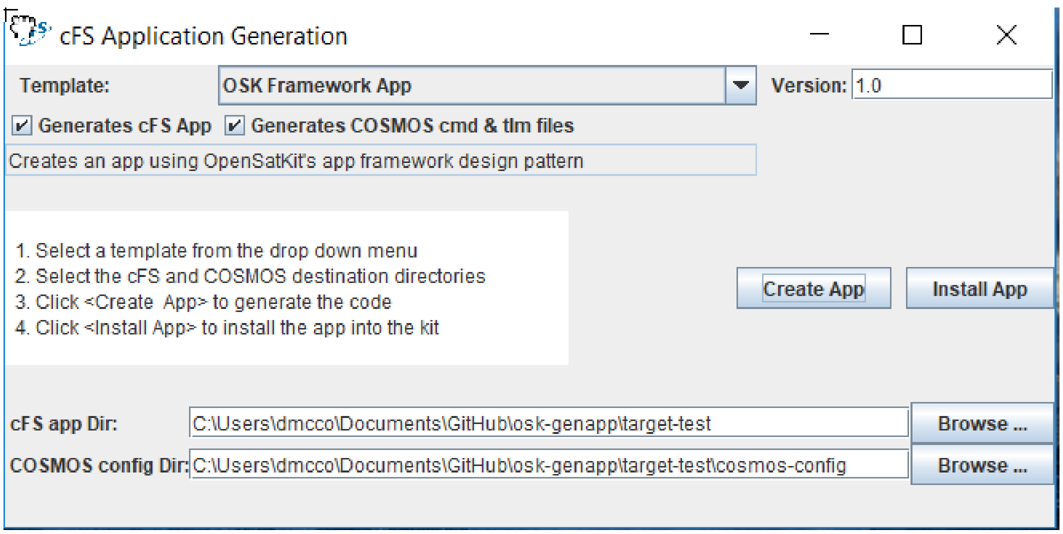


Figure 3-8 – Create Application Tool

Figure 3-9 shows the directories and files that are involved with creating and integrating a new app. The green directories are automatically created and populated by the *Create Application* tool. The green file cosmos/config/tools/cmd_tlm_server.txt is autmatically updated with the new target. The tool inserts the new target name between the <cfs_kit> tags that are under the cFS interface definition line that starts with "INTERFACE CFS_INT" . The yellow highlighted files must be edited and the blue highlighted files provide definitions used by the generated code.

```
cfs
|- apps
|   |- example
|- osk_def
    |- cpu1_cfe_es_startup.scr
    |- targets.cmake
cosmos
|- config
    |- targets
    |   |- EXAMPLE
    |       |- cmd_tlm
    |- tools
       |-cmd_tlm_server.txt
|- lib
    |- message_ids.rb
```

Figure 3-9 – Create Application Directories

Perform the following steps to integrate the *example* application into the kit:

1. Edit cfs/osk_defs/targets.cmake and locate the line starting with SET (TRGT1_APPLIST …) and add "example" to the list of app names).
2. Edit cfs/osk_defs/cpu1_cfe_es_startup.scr and and add the following line.
    a. CFE_APP, /cf/example.so, EXAMPLE_AppMain, EXAMPLE, 90, 16384, 0x0, 0
    b. Note the startup file is processed until the first '!' character is encountered so this line must be before any commented out lines.
3. Note the CreateApp tool generates code that uses existing scheduler table entries and telemetry output entries so those table do not have to be modified. See *Section 3.5 Adding Existing Applications* for a description of how tom modify these tables.
4. Rebuild and install the cFS by issuing the following make commands from the opensatkit/cfs directory
    a. make prep
    b. make install
    c. Under cFE 6.5 people have experienced a cmake failure and a syntax error is report (missing separator). Issuing a "make clean" followed by a "make prep" seems to solve the problem.
5. Install the new app's COSMOS target.
    a. This should be done automatically by the create application tool. Edit cosmos/config/tools/config/tools/cmd_tlm_server/cmd_tlm_server.txt and verify the new app name is located between the <cfs_kit> comment tags under the "INTERFACE CFS_INT udp_cs_interface.rb 127.0.0.1 1234 1235 nil nil 128 nil nil" INTERFACE section.
    b. If you do not see "TARGET EXAMPLE" then manually insert a new line containing "TARGET EXAMPLE" without the quotes. The new line does not have to go between the tags.
    c. You must restart COSMOS for the cmd_tlm_server change to take effect.

6. Follow the instructions in Section 2.1 to start COSMOS and the cFS.
7. Use COSMOS's command and telemetry server to send commands to and receive telemetry to/from the *example* app.

Perform the following steps to integrate the *example* application into the kit:

TBD – Describe apps with tables.

4.5 Adding Existing Applications

The NASA Goddard wesbite (https://cfs.gsfc.nasa.gov/) lists the open source apps released by Goddard. All of these apps come with directory the structure shown in Figure 3-1. The following steps are required to integrate one of these apps. These steps are more involved than the steps peformed when using the *Create Application* tool because the tool automaticly performs some steps and the default message IDs are already defined in the OSK.

1. Copy the application code into the *cfs/apps* dirctory
2. Create a new app target directory in *cosmos/config/targets* and create command and telemetry definition files. See Section 3.2 for details.
3. Note the master copy of the tables defined in the next few steps are located in cfs/osk_defs and prefixed with "cpu1_". When a "make install" is performed it copies the master tables to the "/cf" directory and removes the "cpu1_" prefix. In order to make permanent table changes that take effect modify the tables in cfs/osk_defs.
4. All of the cFS apps send a housekeeping telemetry in response to a specific messsage ID and many apps execute their main loop in response to another message ID. These message IDs are defined in the app's xxx_msgids.h file. These messages are sent from from the Kit Scheduler app. This app uses two tables to manage sending messages on the software bus.
    a. kit_sch_schtbl.xml defines the frequency at which messages are sent. Each scheduler table entry has an index into the message table kit_sch_msgtbl.xml. The following File Manager table entires illustrate how the tables are configured. Message table entry #13 si sent every 3 seconds. The message table entires must be byte swapped for a little endian target and they are specified in decimal. The comment shows the hex(dec)=>Byte swapped hex (dec)
    b. kit_sch_schtbl.xml
        i. <!-- FM -->
        ii. <slot id="1" entry="3" enable="true" frequency="5" offset="0" msg_id="13" />
    c. kit_sch_msgtbl.xml
        i. <!-- FM_SEND_HK_MID 0x188D(6285) => 0x8D18(36120), 0xC000(48152) => 0x00C0(192), 0x0001 => 0x0100(256) -->
        ii. <entry id="13" stream-id="36120" seq-seg="192" length="256" />
5. The app's telemetry packets must be defined in kit_to's table file kit_to_pkttbl.xml in order to be downlinked. The following entry is for the FM app's housekeeping telemetry

packet. The stream_id is the message ID and buffer limit defines how many packets can back up in TO”s queue. 4 is fine.

   a. <!-- FM_HK_TLM_MID 0x088A -->
   b. <entry stream-id="2186" priority="0" reliability="0" buf-limit="4"/>

6. To complete the integration follow the steps in Section 3.4 Creating New Applications.

Additonal applications can be found in the application catalog maintain at http://coreflightsystem.org/. At the moment there is no standard for ensuring compliance with the cFS application conventions so the integration steps may vary.

4.6 Removing Applications

TBD

4.7 Creating Application Unit Test

TBD –

5.0 Implementing a Mission

TBD - Address systems engineering activities. More in depth than a checklist but keep details to a what must be done and references source for how to achieve them.

TBD – Use an example mission. Show end goal and hightlight lifecycle challenges. Some notes…

1. First create a mission concept of operations. This will help flesh out scenarios and functionality required by the FSW to meet the scenarios.
2. Identify existing apps that may meet your needs. The kit doesn't include all of the open source apps but I tried to include the most common. For example, Data Storage and downlink:
   a. If you have ground contacts every X minutes and you're continuously collecting data then the data will needed to be stored. Data Storage will need to be configured to store packets in files. Systems thinking needs to go into what data into which files and how big the files should get.
   b. How is a contact initiated? Stored Command (SC) tables can be used to send commands to configure hardware/software for a time-based event. We often use an absolute time sequence to start relative time sequences that perform a common sequence of commands. SC does not allow parameters to stored commands.
   c. Once the contact is initiated then the files will need to be downlinked which involves CF app (see hurdle #2 below). Another option is to downlink packets. There's a data storage playback app that someone has developed that I could look into. It reads DS files and puts the packets on the software bus so Telemetry Output would send packets to the ground.
   d. Regardless of the downlink method files (which are really CFDP packets) or packets telemetry out table's out table will have to be configured (See hurdle #1 and #3).
3. The cFS does not include hardware interface apps. We typically write one app that collects data and publishes data required by the attitude determination and control (ADC) app. This simplifies the ADC app because it can pend on a single packet. If this isn't doable there are other techniques that keep the ADC app simple.

Some known hurdles:

1. Command Ingest and Telemetry output will need to be replaced. The kit apps are simple UDP apps. Johnson Space Center just released new CI and TO apps that are layered so the same cmd/tlm interface can be used and the data transport library can be swapped at compile time depending upon platform.
2. File transfer: The kit includes a very simple TFTP app. It is not suitable for flight. The cFS CF app is being used on current missions. However it has a steep learning curve and it requires a ground engine. The ASIST and ITOS ground systems (available for government projects) contain a built in CFDP engine. Efforts are underway to create an open source COSMOS CFDP engine.
3. Tables: We do not have a tool do edit tables and generate the binary tables. The Goddard tools are tied to the ground systems we use (ASIST and ITOS).
4. Scripts: Testing and op scenarios are controlled by ground scripts. I've only tested the COSMOS scripting world with the "integration script". A test script framework is planned for Build 1.1.

5.1 Porting to a new platform

TBD

5.2 PiSat

TBD – Convert the following notes:

One thing that is important to know, if you are not aware of it already: The cFS has two different build systems. We have a "legacy" makefile based system, and a newer CMake based system.
I developed the original makefile system, so I am most familiar with that, but many of the newer projects are using the CMake system. For some reason, I don't think the CMake build directory is present in the github link I mentioned above.

I'm pointing that out, because it is fairly easy to cross compile the cFS for the Raspberry Pi using the CMake build system. A little over a year ago, I put together a simple distribution for the cFS along with some instructions for building it on a Raspberry Pi, as well as cross compiling it for the Pi using CMake. I have attached the readme file that explains how to do the cross build for the Pi.
If you follow the directions in that readme file, you should be able to cross compile the cFS for the Pi.

If your team wants to use the traditional make system, it is not that hard to set it up for cross compilation either. I could give you pointers for that, if you would like.

There should not have to be any source changes to make the cFS work on the Raspbian OS.

On the subject of using cFS and Linux for a flight project, there are going to be several things to consider:

1. Do you have any real time requirements that cannot be met with your compute module and Linux? If you do not have any real time control or instrument interface timing issues caused by Linux, then I think it is fine to use for a cubesat.
2. Your team will have to make some decisions about how to implement the Platform Support Package on Linux. I'm not going to get into all of the details here, but you need to figure out how to handle resets, watchdog, memory access, file systems, and a few other items. If your cubesat is fairly simple, then it does not have to be too complicated. But perhaps we can get into more details of what is needed as you make progress.
3. There are a couple of linux issues to figure out, such as how the system will start (just start cFS from rc.local, that's what I do on the PiSat), how to determine if the cFS has crashed and needs to be restarted (do you restart the cFS or the entire Raspberry Pi Linux?)
4. How do you deploy the OS? Do you keep a base linux system on the Compute Module flash and just copy the cross compiled cFS over, or do you try to generate an entire image with the cFS?
    a. Build systems like Yocto Linux, buildroot, or even this nard system look interesting (http://www.arbetsmyra.dyndns.org/nard/). But sticking with the standard Raspbian is always an option too, depending on the size and experience level of your team. I still am looking for a good way to deploy a new copy of the cFS to my Pi-Sat. I have been using the standard Raspbian OS and just building the cFS on the Pi for now.

For the Raspberry Pi flight users, there are a couple of Cubesats that are planning on using a Raspberry Pi compute module, but they are in the early stages of development. There is at least one organization that has flown Raspberry Pi compute modules, but I don't know what operating system they use. I'm sure it is some variant of linux.

All of my work so far with the Raspberry Pi has used the Raspbian OS. The cFS works well on Raspbian, and there are a lot of libraries available for the I/O devices.

Build and run directly on the Pi.
You can also compile on the Pi. Just do the same as the PC:
$ make install
(you might need to remove the build directory first)

Before you compile, there are a couple of flags that must be removed from the build files:
In sample_defs/toolchain-cpu1.cmake, comment out this line:
SET(CMAKE_C_FLAGS_INIT "-m32" CACHE STRING "C Flags required by platform")

And in tools/elf2cfetbl/CMakeLists.txt, comment out this line:
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -m32")

(I tried to get CMake autodetect the system type, but that code does not work yet)

5.2.1 PiSat Attachment

Very basic instructions on getting started with this cFS tree:

================ What's included ================

The followng components were downloaded from the respective project pages
on sourceforge.net:
osal-4.2.1a-release.tar.gz
cFE-6.5.0a-OSS-release.tar.gz
CFS_LIB-Version220.tar.gz
CF-Version221.tar.gz
CS-Version231.tar.gz
DS-Version241.tar.gz
FM-Version242.tar.gz
HK-Version241.tar.gz
HS-Version230a.tar.gz
LC-Version200.tar.gz
MD-Version230.tar.gz
MM-Version240.tar.gz
SBN-Version100.tar.gz
SCH-Version220.tar.gz
SC-Version250.tar.gz

NOTE: This repository is being provided as a convienent distribution for open source software. This repository may not be updated as new versions of the components are released. Please see http://cfs.gsfc.nasa.gov and http://coreflightsystem.org for more information.

==

All of the open source cFS components are included and build with the Cmake rules.
All sample tables compile as well.
Currently not all of the apps are loaded and run when the cFS starts. (sch_lab instead of sch)
But many of the apps load the included default tables,
and the stored command processor will execute the startup RTS.

========== SETUP =============

You will need a linux system with the Gnu Compiler installed along with
32 bit development libraries if you are running on a 64 bit linux distribution.
The 32 bit libraries are installed through the "gcc-multilib"
package on Debian or Ubuntu linux.

You will also need Cmake (sudo apt-get install cmake)

To cross compile for the Raspberry Pi, you will need a Raspberry Pi cross compiler.
Originally I installed the ARM linux cross compiler that was available in the
Ubuntu package repositories. This does not seem to work with the latest version of the
Raspbian OS on the Pi.
Instead I downloaded the compiler from the Raspberry Pi Git repository:
$ git clone https://github.com/raspberrypi/tools

I Downloaded this in my $HOME/Tools directory and set up the path in my .profile:

```
if [ -d "$HOME/Tools/tools/arm-bcm2708/arm-rpi-4.9.3-linux-gnueabihf/bin" ] ; then
  PATH="$HOME/Tools/tools/arm-bcm2708/arm-rpi-4.9.3-linux-gnueabihf/bin:$PATH"
fi
```

To make this work with the build system, you have to edit:
sample_defs/toolchain-rpi-linux.cmake

Change the path in this file to match where you have the compiler:

```
# Specify the cross compiler executables
# Typically these would be installed in a home directory or somewhere
# in /opt.  However in this example the system compiler is used.
SET(CMAKE_C_COMPILER          "/home/alan/Tools/tools/arm-bcm2708/arm-rpi-4.9.3-linux-
gnueabihf/bin/arm-linux-gnueabihf-gcc")
SET(CMAKE_CXX_COMPILER        "/home/alan/Tools/tools/arm-bcm2708/arm-rpi-4.9.3-linux-
gnueabihf/bin/arm-linux-gnueabihf-g++")
```

====== COMPILE and RUN ======================

NOTE: With the Cmake build system, the "build" or "build-rpi" directories are generated by the tools. It will not hurt to remove them to start a clean build.

1. Compile the current cFS configuration to run on x86 Linux (32 bit)
 $ make install
 The compiled executables will be located in the build/exe/cpu1 directory.
 To run it:
 $ cd build/exe/cpu1
 $ sudo ./core-cpu1
 (Sudo is necessary for the linux real time schedule policy and the message queues)
 When the cFE is running, you will see the event messages come out on the console. Hit "control-c" to stop it.

 *************** 64 BIT NOTE **************
 If you are running on a 64 bit linux system, you will have to make sure the -m32 flag is
 enabled in the following files:
 1. sample_defs/toolchain-cpu1.cmake
 2. tools/elf2cfetbl/CMakelists.txt
 ****

2. Cross compile the current cFS configuration for the Raspberry Pi.
 If you have the cross compiler setup as described above, you just need to run:
 $ make SIMULATION=rpi-linux O=build-rpi install
 (In this case the SIMULATION flag is describing an alternate toolchain)
 The compiled executables will be located in the build-rpi/exe/cpu1 directory.

 To run it, you will need to transfer the build-rpi/exe/cpu1 directory to the Pi,
 and run as you would on the PC:

 $ cd cpu1
 $ sudo ./core-cpu1

 note: sudo is used for two things:
 1. the Real time scheduler policy on Linux
 2. The cFE/OSAL use Posix Message Queues, which by default
 sizes the message queue sizes too small.

 The cFE and cFS are now running. The CI_LAB app is waiting for commands
 on udp port 1234
 The TO_LAB app will send telemetry to the commanded host

 To stop the cFS, hit control-c.

3. Build and run directly on the Pi.
 You can also compile on the Pi. Just do the same as the PC:
 $ make install
 (you might need to remove the build directory first)

 Before you compile, there are a couple of flags that must be removed from the build files:
 In sample_defs/toolchain-cpu1.cmake, comment out this line:

SET(CMAKE_C_FLAGS_INIT "-m32" CACHE STRING "C Flags required by platform")

And in tools/elf2cfetbl/CMakeLists.txt, comment out this line:
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -m32")

(I tried to get CMake autodetect the system type, but that code does not work yet)

===== Commands and Telemetry ==========

The cFE open source release includes a simple ground system written in Python/Qt
We are working on replacing it with a more full featured system:
(http://cosmosrb.com)

How to send commands and receive telemetry:

1. Open a terminal
2. Switch to the cFS ground system directory:
   $ cd tools/cFS-GroundSystem
3. Start the menu:
   $ python GroundSystem.py
4. Click on the "Start Command System" button
5. On the Command System Main Page, Click on the "ES No-Op" button.
   You should see an event message appear in the terminal where the cFS is running.
6. Click on the "Enable Tlm" button, a new window will pop up
7. In the "dest_IP" Parameter row, enter the following in the "Input" box:
   127.0.0.1
   Then click the "Send" button. Telemetry should now be enabled and sent to the ground system. You should see an event in the cFS terminal.
8. Back on the Ground System Main Window, click "Start Telemetry System". You should start to see telemetry packets counting up in the Telemetry System Page.
9. In the "Event Messages" row, click the "Display Page" button.
10. Now, go back to the command page and send a few no-ops. You should see the events being received in the event message window.

5.3 Configuring the cFE

Startip log & messages
Time

5.4 Creating your application suite

Ops concepts

5.5 Development process

TBD

5.6 Ground system

TBD

5.7 Systems Topics

6.0 Test Framework

TBD – Describe test framework from app unit, system integration, and possibly build verification depending on how the cFS ground-system independent testing goes

7.0 Kit Design and Maintenance

This section describes the kit's design. It provides enough information for someone to maintain and expand the kit. If this section gets too large it will be split into a separate document. It is included with the user's guide because most users are developers and they may want more information even if they're not maintaining the kit.

7.1 COSMOS Configuration

The COSMOS directory structure is shown in Figure TBD.

```
cosmos
|- cfs-kit
   |- docs
   |- file_server
      |- tables
   |- tools
      |- create-app
      |- perf-monitor
|- config
   |- data
   |- system
   |- targets
      ...
      |- cfs-kit
         |- cmd_tlm
         |- lib
         |- screens
      ...
   |- tools
|     |-cmd_tlm_server.txt
|- lib
|- outputs
|- procedures
   |- kit_test
   |- kit_utils
|- scripts
|- tools
```

Ruby search path
Cosmos caching

7.1.1 Ruby Gems

7.2 OSK COSMOS Design

TBD – Add module/class diagram

Describe design patterns like file_xfer that allows windows to remain open across switch between local and pisat and still work.

TBD Conventions

1. Naming conventions. Screens, scripts, etc
2. Script usage
3. Filer server conventions
4. Spawn vs cosmos.run_process
5. Flight directories are always full path. Ground directories are relative to COSMOS

7.3 42 Configuration

42 is customized for OSK, therefore care must be taken when integrating a 42 release. Some of the modifications follow the 42 extension guidelines in Section 3.5 in FswModels.pdf. The following files must merged or added back to 42:

- Merged:
    a. 42/Include/42defines.h – EXTERNAL_FSW controller definition
    b. 42/Source/42init.c - EXTERNAL_FSW controller string decoding
    c. 42/Source/42fsw.c – Add f42_comm interface
    d. MakeFile – Add f42_comm object
- Added:
    a. 42/Includes/f42_comm.h
    b. 42/Source/f42_comm.c

I like to use WinMerge (or a similar tool with difference/merging feature) to perform the following 42 file merge process:

c. Save the above files to a temporary location
d. Delete the entire 42 repository and install the latest 42 release in its place
e.

7.4 Kit Application Design

TBD – Provide basic description and reference kit app design document

7.5 cFS Component Releases

This sections outlines the steps needed to update a particular component of the starter kit.

7.5.1 cFE

Tools
Copy docs
CFE_ES_SYSTEM_LOG_SIZE to 16384

Appendix A - Acronyms

API……………………………………….……………………………………Application Programming Interface
cFE...Core Flight Executive
C&DH .... Command and Data Handling
CCSDS………………………………………………………….. Consultative Committee for Space Data Systems
cFS………………………………………………………………………………………Core Flight Software System
CM .... Configuration Management
CMD...Command
COTS .... Commercial Off-The-Shelf
CPM .... CFS Performance Monitor
CPU .... Central Processing Unit
DCR .... Discrepancy/Change Request
EDAC .... Error Detection and Correction
EDS...Electronic Data Sheet
EEPROM .... Electrically-erasable Programmable Read-Only Memory
ES .... Executive Services
ETU .... Engineering Test Unit
EVS-- ....Event Services
FC .... Function Code
FDC.. Failure Detection and Correction
FSB .... Flight Software Branch
FSW ....Flight Software
HW .... Hardware
ICD..Interface Control Document
I&T .... Integration & Test
MET ....Mission Elapsed Time
OS..Operating System
OSAL……………………………………………………………………….……Operating System Abstraction Layer
PID ....Pipe Identifier
RTOS .... Real-Time Operating System
SB ....Software Bus Services
STCF...Spacecraft Time Correlation Factor
T&C .... Telemetry and Command
TAI .... International Atomic Time
TBD..To Be Determined

TBL .... Table Services

TIME .... Time Services

TLM .... Telemetry

URL .... Universal Resource Locator

UTC .... Coordinated Universal Time

UTF…………………………………………………………………………………………Unit Test Framework

VDD .... Version Description Document

VM .... Virtual Machine

Appendix B – Online Resources

B.1 First Time Kit Installation

The current kit installation script is designed to run on Ubuntu. Due to limited resources we wanted to keep the installation simple, robust, and easy to maintain. A single platform significantly lowers the resources required to verify the installation.

Perform the following steps to install the kit:

1. Create an Ubuntu host platform. A virtual machine is the typical solution. The osboxes.org website provides freely available Ubuntu images for Virtual Box and VMWare.
    a. TBR – Check if auto login allows cFS to be launched without a prompt so telemetry can be enabled.
2. From within the Ubuntu platform go to https://opensatkit.github.io and you'll see the following information.

Install (Ubuntu only)

```
$ bash <(\wget -qO- https://raw.githubuserc
ontent.com/OpenSatKit/OpenSatKit/vendor/ins
tall.sh)_
```

3. Open a terminal window on your Ubuntu platform. Copy the bash line from the website into your terminal window and the run the shell script. This script will take from 30-60 minutes to complete depending upon your internet connection and host machine performance. There are some prompts that must be answered.
4. After the script successfully runs COSMOS is automatically launched. Refer to Section 2 for how to use the kit features.

In order to use the PiSat (see TBD for obtaining a PiSat) port forwarding must be enabled in the VM.

B.2 Updating the Kit

TBD

B.3 COSMOS Resources

http://cosmosrb.com

B.4 42 Resources

TBD

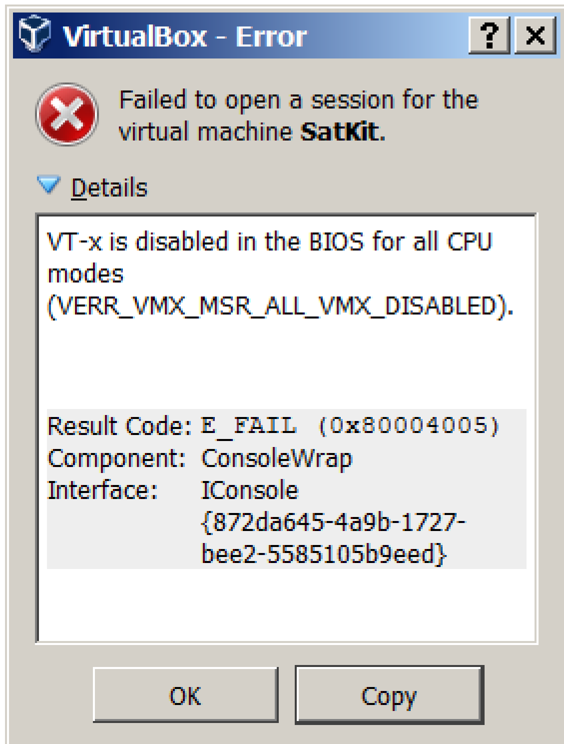
B.5 PiSat

TBD

Appendix C – User FAQs

C.1 Installation Issues

1. When I try to start a virtual machine either after I importing a .ova file or when I try to run any virtual machine, I get an error such as



    a. This particular error is due to the virtual technology extensions (VT-x) being disabled in your BIOS. You can enable VT-x by performing the following steps
        i. Reboot your PC. As soon as the first logo appears immediately press the F2 key
        ii. Locate the VT-x settings (menu system specific to your bios) and change it to enabled

2. When I try to update COSMOS by running "bundle update cosmos" from my COSMOS project folder I get a 'permission denied' error in a 'dot' folder such as .rbenv or .gem when a gem is being installed. This typically occurs in a VM. It is best not to install the gem as root so the easiest solution is to change the file permissions for the particular 'dot' directory and all of its

children. The first line below shows the general format and the second line is a specific example.

a. sudo chown -R username:group ~/.xxx
b. sudo chown -R vagrant:vagrant ~/.rbenv

3. I don't have all of the tools to build and run the kit. The safest way to install the kit is to use the shell script on github. Here are some specific error and causes that may be helpful:

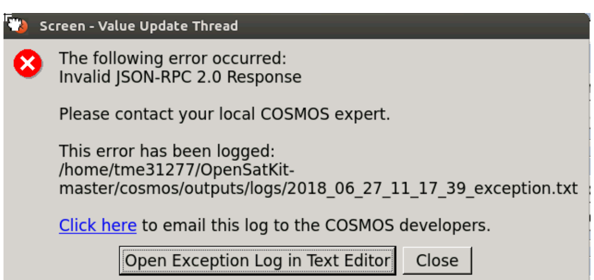
   a. cmake not found
      i. sudo apt-get install cmake
   b. cmake prep fails due to test program not being compiled.
      i. sudo apt-get install gcc-multilib
   c. cmake build fails during OSAL with obscure error messages
      i. sudo apt-get install –y curl
   d. Java runtime not found when trying to run java-based kit tools
      i. sudo apt-get install default-jre
   e. When a try to start the cFS I get a xfce error
      i. sudo apt install xfce4-terminal

C.2 Ubuntu Issues

1. The COSMOS menu bar selections are missing.
   a. They should be displayed in the Ubuntu main window for the COSMOS window. This is the default Ubuntu 16.04 setting.
   b. To change the menu bar display select the Ubuntu settings icon (wheel cog) in the upper right. Then select Appearance->Beahvior-> "Show the menu for a window" and choose "In the window title bar".

C.3 COSMOS-cFS Connections Issues

1. When I start the cFS I get the following COSMOS dialogue box



   a. This problem is addressed in the COSMOS installation instructions: http://cosmosrb.com/docs/installation/. "The http_proxy environment variable can cause problems. Make sure you also have a no_proxy variable for localhost, something like no_proxy="127.0.0.1, localhost". "

2. When I start the cFS I get an error box stating failed to establish connection.
    a. Press the 'Enable Telemetry" button on the main screen.
    b. If this doesn't work then try restarting the COSMOS Command & Telemetry Server. The cFS can remain running when you do this.

3. After I start the cFS it seems to run fine, and I see event messages in the terminal window, but I don't see any telemetry
    a. Look at cFS startup messages. If there are kit_ci and/or kit_to socket bind error 98 then a cFS process is still running
    b. Open a new terminal window (ctrl-alt-t on an Ubuntu VM)
    c. Issue a 'pgrep core' command. If more than one process ID shows up then you have an orphan cFS running.
    d. Issue a sudo kill 'xxxxx' to end the old process where xxxxx is the process ID

C.4 COSMOS

1. After I change an Embedded Ruby script I don't see it take effect.
    a. The results of the ERB processing are cached. You either have to modify the file, or delete the cache from the outputs/tmp folder.

C.5 cFS

1. When I start the cFS I see a bunch of messages. Should I be concerned?
    a. TBD
2. What is the "fly wheel" message all about?
    a. The cFE Time service tries to remain synchronized with a 1Hz time at tone message. If Time can't verify the synchronization, it declares it's in "fly wheel" mode and is no longer synchronized. The timing on Ubuntu and especially within a VM is not as deterministic as cFE Time expects so it often enters/exits "fly wheel" mode. See the cFE Time documentation for more details.
3. I modified ../cf/cfe_es_startup.scr and my changes disappeared.
    a. When you build the cFS with cmake and then do a 'cmake install' it copies the cfe_es_startup.scr from the root cmake directory into the target CPU's boot directory. Therefore if you modify the target startup script it will get overwritten when you install a new build.
4. When I start the cFS a terminal window opens and the cFS looks like it starts but nothing updates.
    a. First confirm it's not a connection issue. See FAQ C.1.1.
    b. Look through the startup messages and if you see messages stating tables were not loaded then the problem is most likely that the default tables in the /cf directory were

deleted during the cFS build process. These tables include the scheduler tables and the telemetry output table.

C.6 42

1. The 42 graphics is extremely slow in my VM
    a. In the VM display settings turn off 3D acceleration
    b. In 42/InOut/Inp_Graphics.txt under the “CAM Show Menu” category set “Shadows” to FALSE

C.7 JSON Files

1. Here are some common JSON parser formatting mistakes. The exact output of the ground and flight software JSON parsers are not identical but they should be similar to the following errors.
    a. Expecting 'STRING' - You probably have an extra comma at the end of your collection. Something like { "a": "b", }
    b. Expecting 'STRING', 'NUMBER', 'NULL', 'TRUE', 'FALSE', '{', '[' - You probably have an extra comma at the end of your list. Something like: ["a", "b",]
    c. Enclosing your collection keys in quotes. Proper format for a collection is { "key": "value" }
    d. Make sure you follow JSON's syntax properly. For example, always use double quotes, always define your keys in quotes, and remove all callback functions.

Appendix D – Naming Conventions

D.1 Command & Telemetry Database

FM HK_TLM_PKT
State processor endian at top of packet and don't repeat for appended data parameters
I followed the command mnemonics abbreviations in the HTML. However I made CAPS and used underscores rather than the CamelBack.

I used "write" when writing info to a file and "send" when sending a telemetry packet with info.

Descriptive comments. Start command descriptions with a verb. No period at the end of single sentence comments.

Whole names with no underscores
BITMASK
FILENAME

D.2 Abbreviations

ACT	Active
ADDR	Address
ADJ	Adjust
APP	Application
ATP	Absolute Time Processor
ATS	Absolute Time Sequence
BIN	Binary
BUF	Buffer
CLR	Clear
CMD	Command
CNT	Count
CONT	Continue
CRC	Cyclic Redundancy Check
CTR(S)	Counter(s)
DEST	Destination
DIAG	Diagnostic
DIS	Disable
ENA	Enable
ERR	Error
EXE	Executing
EVT	Event
FLTR	Filter
FMT	Format
IDX	Index
MET	Mission Elapsed Timer
MON	Monitor

MSG	Message
NUM	Number
OVFL	Overflow
PARAM(S)	Parameter(s)
RECV	Receive
REG	Register, Registry
REV	Revision
RST	Reset
RTS	Relative Time Sequence
SEQ	Sequence
STATS	Statistics
STCF	Spacecraft Time Correlation Factor
SUB	Subtract
SYS	System
TBD	To Be Determined
TLM	Telemetry
VAL	Value, Validate
VER	Version
WHL	Wheel