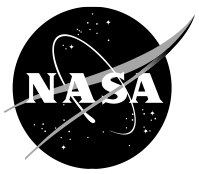
Software User's Guide
for the
Core Flight System Test Framework Tool

Engineering Directorate
Software, Robotics, and Simulation Division

Availability:
NASA & NASA contractor employees as required

March 2022
Baseline



National Aeronautics and
Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Verify this is the correct version before use

Change Record

Revision	*Date*	*Originator*	*Description*
N/A	Sep 2021	Tam Ngo & Tao Wu	Initial draft
N/A	Mar 2022	Tam Ngo	Baseline

Table of Contents

List of Tables

List of Figures

1 INTRODUCTION

1.1 Scope

This Software User's Guide is for the Core Flight System (cFS) Test Framework (CTF) ground software tool. From here, the tool will be referred to as the CTF.

1.2 Purpose

This document describes how to install, configure, build, execute and troubleshoot the CTF within the context of a cFS tool. The tool was developed specifically as part of the cFS ecosystem, and hence, it provides cFS-specific interfaces to interact with cFS instances.

1.3 Audience

The intended audience of this document are the cFS software developers and testers who integrate CTF into their cFS-based software system. It is assumed that the developers are familiar with the general infrastructure of the cFS and its ecosystem as well as the general build and run of cFS applications and libraries.

1.4 Document Status and Schedule

CTF Software User's Guide is part of the documentation that comes with the software release of the CTF software tool.

2 RELATED DOCUMENTATION

2.1 Applicable Documents

The following documents, of the exact issue and revision shown, form a part of this Software User's Guide to the extent specified herein.

Table 2-1: Applicable Documents

Document Number	***Document Title***	***Revision / Release Date***
NPR 7150.2	NASA Software Engineering Procedural Requirements	Rev C / Aug 2019
EA-WI-025	GFE Flight Project Software and Firmware Development	Rev D / Sep 2013
JSC-61949	Advanced Exploration Systems (AES) Core Flight Software (cFS) Software Development Plan	Rev C / Mar 2021

2.2 Reference Documents

The following documents are reference documents utilized in the development of this Software User's Guide. These documents do not form a part of this document and are not controlled by their reference herein.

Table 2-2: Reference Documents

Document Number	*Document Title*	*Revision / Release Date*
GSFC 582-2007-001	cFE Application Developer's Guide	Rel. 5.4 / Sep 2014
GSFC 582-2008-012	cFS Deployment Guide	Rel. 3.0 / Sep 2014
N/A	CTF Software Design Document	Baseline / Mar 2022
N/A	CTF Test Plan and Procedures	Baseline / Mar 2022

3 OVERVIEW

The Core Flight System Test Framework (CTF) is a ground software tool that provides cFS projects with the capability to develop and run automated verification tests. The CTF tool parses and executes JSON test scripts containing test instructions, while logging and reporting the test results. CTF utilizes a plugin-based architecture to allow the users to extend CTF with new test instructions, external interfaces, and custom functionalities. CTF comes with a user-friendly, graphical editor (GUI) that provides context-sensitive, auto-suggestions about a cFS system's command and telemetry definitions.

4 INSTALLATION

To get started, clone the CTF repository using the following command:

```
$ git clone https://github.com/nasa/ctf
```

It creates a directory named ctf. Its file structure is listed as below.

```
├── activate_ctf_env.sh
├── ctf (executable)
├── ReadMe.md
├── requirements.txt
├── run_editor.sh
├── run_tests.sh
├── setup_ctf_env.sh
├── configs/
├── docs/
├── external/
├── lib/
├── plugins/
├── functional_tests/
├── example_scripts/
├── vv_tests/
├── unit_tests/
└── tools/
    └── ctf_ui/
    └── schema_validator/
```

The **`configs`** folder contains a few sample CTF configuration INI files.

The **`docs`** folder contains the CTF document files, including this document.

The `external` folder contains a sample CFS workspace zip file.

The `lib` and `plugins` folders contain the Python source files for the CTF core components and plugins. The details can be obtained from CTF Software Design Document.

The `functional_tests`, `vv_tests`, `vv_tests` folders contain the sample CTF json test scripts files.

The `unit_tests` folder contains Python unit test files for CTF source code files.

The `tools` folder contains the source files for CTF Editor and schema validator tool.

4.1 CTF Prerequisites

CTF has been developed and tested on CentOS 7 Linux and requires Python 3.x. The CTF Editor requires NodeJS / NPM. **Table 4-1** lists all the 3rd-party software dependencies required by CTF. They must be installed for the tool to work properly.

Table 4-1: 3rd-Party Software Dependencies

4.2 Installation of 3rd-Party Packages

There are two methods to install package dependencies:

1. An Anaconda environment setup script is provided to install all OS/Python dependencies into a self-contained Anaconda environment. This method does not require superuser privileges. However, the CTF Python environment must be activated prior to running CTF.
2. A PIP `requirements.txt` file is provided to install all Python 3 CTF dependencies. OS dependencies need to be installed manually and may require superuser privileges. This method provides the most light-weight dependency installation, but more setup details are involved than the Anaconda.

4.2.1 Installation with the Anaconda

The `setup_ctf_env.sh` script will setup an Anaconda 3 environment, which contains Python3, along with the Python components identified in the `requirements.txt` file and NodeJS/NPM.

1. To set up the CTF environment, execute the command

   ```
   $ source setup_ctf_env.sh
   ```

   Note that this may take several minutes depending on your network connection and will consume about 5Gb of disk space for Anaconda 3 and NPM dependencies.

2. After the initial setup, to activate the Anaconda environment, execute the command

   ```
   $ source activate_ctf_env.sh
   ```

   The console prompt will change to

   ```
   (pythonEnv3) [user@user-centos-vm ctf]$
   ```

If the Anaconda environment is corrupted, the environment can be reinstalled by executing the command

```
$ source setup_ctf_env.sh -u
```

4.2.2 Installation with PIP

Follow the steps below to perform package installation with PIP.

1. First, install Python3. On the CentOS, execute the following command

   ```
   $ sudo yum install python3-devel python3 python3-pip
   ```

2. Then install NodeJS/NPM, visit https://nodejs.org/en/ and install Node version >= 10.12.0 (tested with v12.5.0). NPM will be included in the installation.

3. With NPM installed, the editor and its dependencies can be installed by executing the following command

   ```
   $ cd tools/ctf_ui && npm install
   ```

   Note that this may take several minutes depending on your network connection and consume about 0.5Gb of disk space.

4. With Python 3 installed, the PIP dependencies can be installed by executing the following command

   ```
   $ pip install -r requirements.txt
   ```

   Note that to ensure that dependencies are installed to a PIP user for easy updates later, the users can execute the above command with `--user` option added.

5. Also, to generate documentation, install Doxygen. On the CentOS, execute the following command

   ```
   $ sudo yum install doxygen
   ```

4.3 Installation of `sample_cfs_workspace`

Being a cFS test tool, CTF needs to interact with a running cFS system. A project would typically create a repository/workspace to develop, test, build and run its cFS system. The **`sample_cfs_workspace`** is a sample of such system. For a recommended cFS workspace, see https://github.com/nasa/cfs.

Note that in a git repository for a cFS workspace, CTF repo can be git-submoduled inside that cFS repository, under the **tools** directory, to track CTF version used by the project.

To setup this workspace for CTF checkout, follow the steps below.

1. First, from a terminal window, navigate to the `external` directory and uncompress the **`sample_cfs_workspace.tgz`** to a directory outside of the CTF directory, such as the user's home directory.

   ```
   $ tar -C ~/ -xvf sample_cfs_workspace.tgz
   ```

Verify that this is the correct version before use

2. Next, navigate to that directory and build cFS to ensure all cFS dependencies are installed.

```
$ cd ~/sample_cfs_workspace
$ make prep; make; make install
```

   Note that on a fresh CentOS 7 installation, the following dependencies are required to build a cFS project

```
$ sudo yum install cmake glibc-devel.i686 libgcc.i686
```

3. Ensure the cFS instance can be started by executing

```
$ cd build/exe/lx1
$ ./core-lx1
```

   Let cFS runs for few minutes and enter **<Ctrl-C>** to terminate the execution.

   Now that it is verified that a cFS instance can be built and executed properly, the users can proceed to configure CTF to run test scripts against the cFS system in the **sample_cfs_workspace**.

5 CONFIGURATION

In this section, it describes how to configure CTF and **sample_cfs_workspace**. As a cFS test tool, CTF needs to interact with cFS instances. The CTF configuration file provides the information to identify/execute cFS. The **sample_cfs_workspace** configuration is for CTF Editor (GUI) to build/run test scripts. If the users want to run the tests without the GUI, the **sample_cfs_workspace** configuration could be skipped.

5.1 CTF Configuration

As stated in **Section 4**, the CTF file structure contains a **configs** directory, which includes a few samples of CTF configuration file. A CTF configuration file contains configuration options for the CTF engine. To run CTF with a selected configuration, add the **--config_file <path-to-config-file>** to the command line argument when starting CTF. If the **--config_file** argument is skipped, CTF will use the **ctf/configs/default_config.ini** file.

In addition, each plugin can define one (or more) sections with specific configuration values as follows

```
[some_plugin_config]
some_field = true/false
some_other_field = xyz
```

Example configuration files are provided under **configs/examples**. It is recommended to create different configurations for different use cases, such as a git Continuous Integration config, a Test Authoring/Debugging config, etc.

The below lists a few important configuration items. Please note that the users only need to modify the first item, **workspace_dir**, if using the **default_config.ini** for the cFS system in the **sample_cfs_workspace**.

1. `workspace_dir = ~/sample_cfs_workspace`

This is the cFS system home directory. All cFS paths will be relative to the **workspace_dir**. If it is not configured properly, CTF could not start cFS instances.

2. `build_cfs = true`
   If it is set to True, CTF will recompile/build the cFS instances before executing them.
3. `cfs_exe = core-lx1`
   It is the name of a cFS instance. If a cFS instance other than **sample_cfs_workspace** is used, the name needs to be updated.
4. `CCSDS_data_dir = ${cfs:workspace_dir}/ccdd/json`
   It is the directory of the CCSDS command and telemetry definitions files in JSON format. These definitions are used for CTF to send command and receive telemetry messages, as well as by the editor for auto-suggestion features.
5. `evs_messages_clear_after_time = 5`
   It is the setting for the CFE EVS message validations. Setting this value to X will allow CTF to validate events received within the past X time-units and thereon. Packets received prior to X time-units are not accepted for validation.

5.2 sample_cfs_workspace Configuration

The CTF tool contains a minimal cFS workspace, **external/sample_cfs_workspace.tgz**, for CTF evaluation purposes. After the installation described in **Section 4.3**, navigate to the **sample_cfs_workspace/ctf_tests** directory. The workspace configuration file is called **editor_workspace.json**. Note that this configuration is specifically for the CTF Editor, defining the locations of various components used by the Editor. If the CTF users opt to run the test scripts without the editor, this step could be skipped.

Below is a sample content of a workspace configuration.

```
{
    "workspace_file_description": "CTF Workspace File - Configures the editor
     directories with the appropriate CTF Project Dir, Scripts Dir, CTF Executable,
     Plugins, and CCDD",
    "projectDir_notes": "CTF Working Directory - Directory where scripts, results, and
     configs are placed)",
    "projectDir": "./",
    "scriptsDir_notes": "CTF Scripts Directory - Directory where scripts are. Files
     within are shown in the scripts list of the editor",
    "scriptsDir": "../tools/ctf/scripts/cfe_6_7_tests/",
    "ctfExecutable_notes": "CTF Executable - CTF executable located at the root
     directory of CTF repo",
    "ctfExecutable": "../tools/ctf/ctf",
    "pluginDir_notes": "CTF Plugin Dir - Directory containing plugin information for
     editor to ingest. Defaults provided in CTF Repo",
    "pluginDir": "../tools/ctf/plugins/info/",
    "ccddJsonDir_notes": "CCDD JSON Dir - Directory containing JSON CCSDS files
     exported from CCDD. Used by the editor for autosuggestion features",
    "ccddJsonDir": "../ccdd/json/"
}
```

The details of how to run CTF Editor are described in **Section 6.3**.

6 USAGE

This section describes how to activate the CTF environment, run the test scripts, run the CTF Editor, as well as run the unit tests and static analysis.

6.1 How to Activate CTF Environment

1. To activate the CTF environment after installing Anaconda, navigate to CTF directory from a terminal window, and run the command:

   ```
   $ source activate_ctf_env.sh
   ```

   The users will be prompted to enter the conda environment location. (This is only needed if a different directory was specified in the setup script). If successful, the `(pythonEnv3)` prompt will be added to the terminal prompt. From there, the users can run `ctf` or `ctf_editor` from the `ctf` directory.

Note that the Anaconda environment only needs to be installed once. But when a terminal is open for a new session, the CTF environment needs to be re-activated.

6.2 How to Run the Provided CTF Test Scripts

The CTF Test Plan and Procedures (STP) contains instructions on how to run various CTF tests provided for purpose of unit testing, functional testing and code coverage, verification testing and static code analysis.

6.3 How to Run CTF Editor

The CTF tool provides a CTF Editor to assist in the creation, modification, and running of the test scripts. It can be obtained from the CTF repository under the directory of `tools/ctf_ui`. CTF and CTF Editor are independent applications, the users can run CTF without the Editor and vice versa.

6.3.1 Getting Started

To launch the editor, make sure that the CTF environment is activated as described in **Section 6.1**. If not, execute the following command at the root directory of the CTF repo.

```
$ source activate_ctf_env.sh
```

After activating the environment, executing the following script to launch the editor.

```
$ ./run_editor.sh
```

Note that if errors are seen related to the `chrome-sandbox` permissions, execute the following commands:

```
$ sudo chown root /path/to/chrome-sandbox
$ sudo chmod 4755 /path/to/chrome-sandbox
```

6.3.2 CTF Editor Layout

Figure 6-1 depicts the layout of the CTF Editor's UI components.

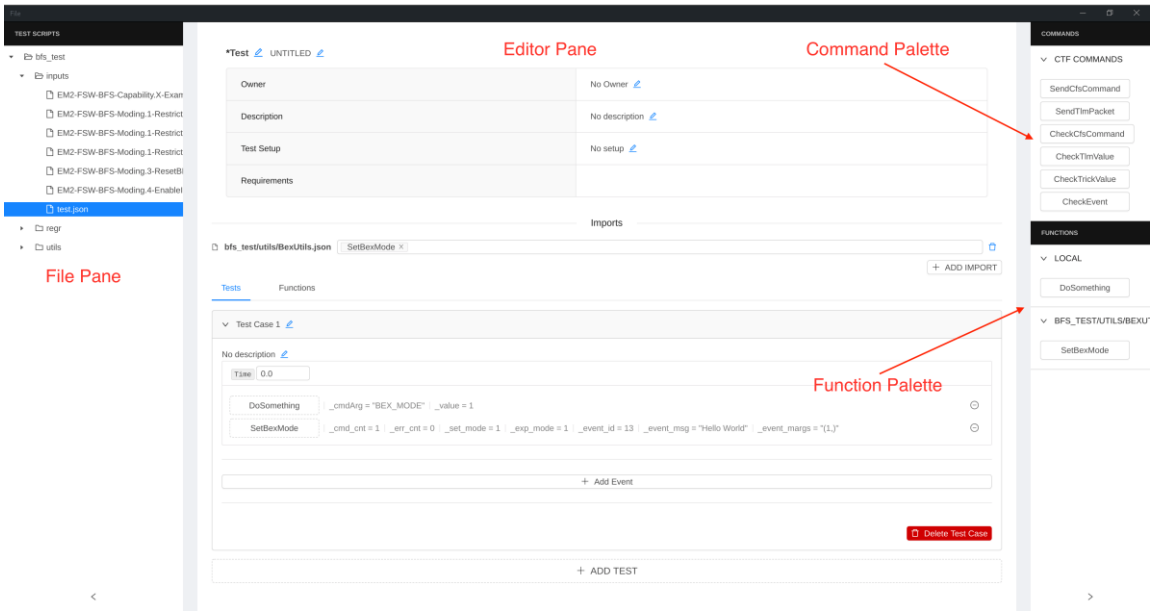


Figure 6-1: CTF Editor Layout

- File Pane - Shows the directories and JSON scripts in the provided project workspace.
- Editor Pane - Shows the current script being edited.
- Command Palette - Contains all the commands that can be dragged into the current script.
- Function Palette - Contains all the functions that can be dragged into the current script.

6.3.3 Loading a Workspace

From the menu at top left corner, click **`File->Open Workspace`** to specify the workspace. It will show a pop-up window from which the users can navigate the file system to select the workspace configure file.

The workspace includes the base path of JSON scripts, the path to JSON files containing the available CTF commands, and a path to the Command and Data message definitions in JSON schema in order to offer the autocompletion of the data fields.

6.3.4 Creating a New Test Script

To create a new test script, from the menu click **`File->New Test Script`**. This will create an empty (unsaved) test script. The users can modify any of the script header's properties by clicking the `edit` icon. After modifying these fields, save the script by entering **`<CTRL-s>`** or clicking **`File->Save Test Script`**. The users will be prompted for a location and a name for the new script file.

6.3.5 Building a Test Case

To build a test case, click the '**`+ Add Test`**' icon. This will create a collapsed test case with the name **`Untitled Test`**. The title can be modified along with the test description. Remember to save the test case periodically so that updates will not be lost.

With the test case created, the users can now add the test instructions. To do that, drag instructions from the top right **`Commands`** pane into the test case. After dropping an instruction into the test, click that instruction button to modify the instruction arguments. Most argument fields will provide autocomplete functionality for field types CTF Editor knows (MIDs, CCs, Telemetry Variables, etc...). When finished with that test instruction, click anywhere to hide the arguments pane. And remember to save periodically!

6.3.6 Modifying an Existing Test Script

To modify an existing test script, choose the existing script from the left pane of CTF Editor. The script will be editable in the main editor pane, and any change can be saved by typing **`<CTRL-s>`** or clicking **`File->Save Test Script`**.

It is recommended that the users run the modified test script through a diff tool to ensure that the changes introduced by CTF are, in fact, what the users expect the changes to be.

6.3.7 Creating a Test Function

The test functions allow the users to share a sequence of commands among various test cases or test scripts. The functions can be defined by switching to the **`Functions`** tab.

A test function defines a set of parameters that can be used throughout the function scope. **Figure 6-2** below shows a function definition **`DoSomething`** which receives 2 parameters, **`_cmdArg`** and **`_value`**. The test instructions in that function can use those parameters wherever suitable.

Figure 6-2: An Example of a Test Function

6.3.8 Importing a Test Function

Test functions can be imported from other JSON files. To import a test function, click '**`+ Add Import`**' and select the JSON file that contains the functions of interest. Next, select the functions to import from that file. The function will now be available as shown in the bottom right **`Functions`** palette.

An example of an imported test function is shown below in **Figure 6-3**.

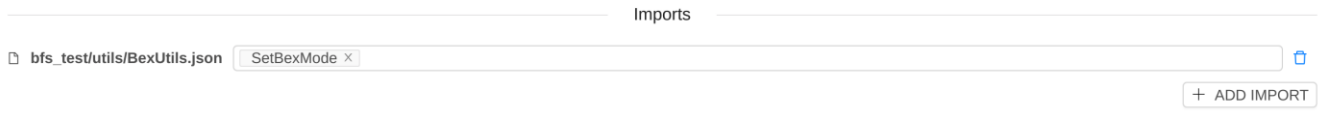


Figure 6-3: An Example of an Imported Test Function

6.3.9 Adding a Test Function to a Test Case

To add a test function to a test case (or within another function), drag the function from the **`Functions`** palette at the bottom-right corner into the test script. Next, click the new function to specify its arguments. An example of two functions calls is shown below in **Figure 6-4**.

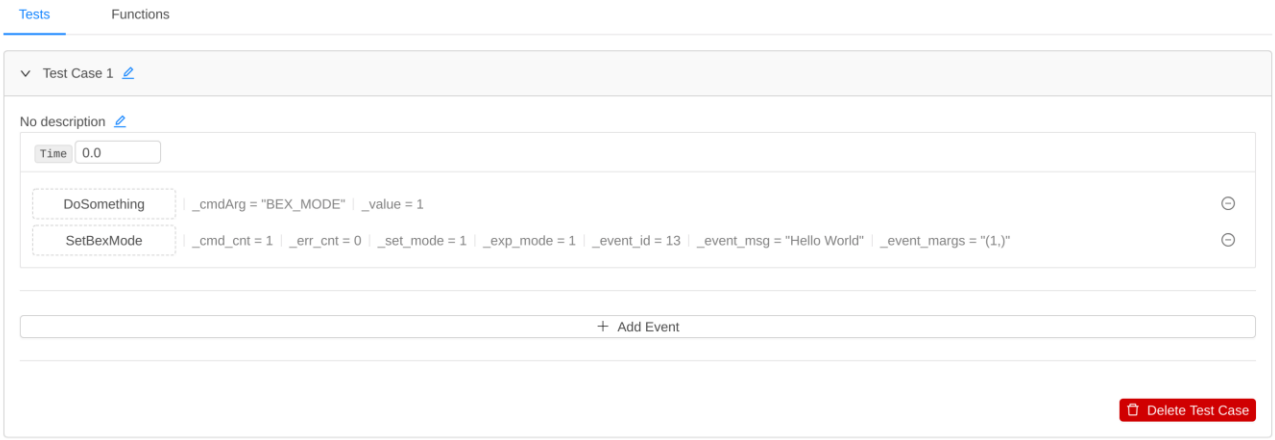


Figure 6-4: An Example of an Added Function

6.3.10 Running the Test Scripts

To run a test script, navigate through the folders/files in the **`File`** pane, select the test script file, then right click the file and choose **`Run(Default Config)`** or **`Run(Custom Config)`** as shown in **Figure 6-5** below.

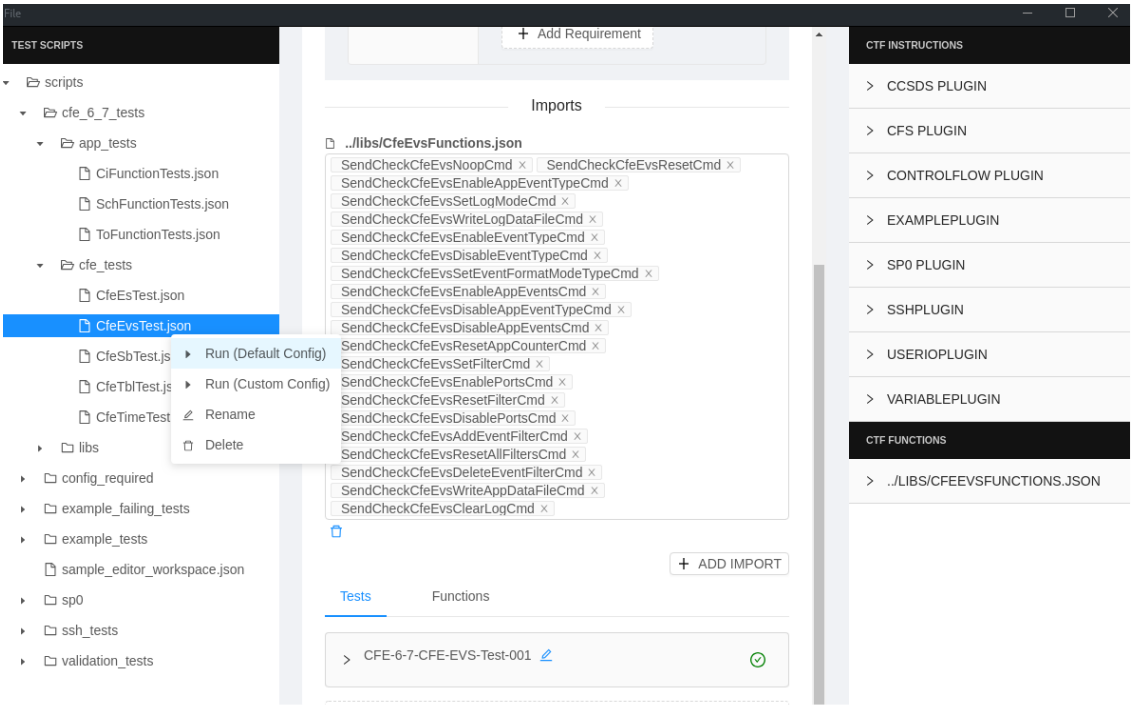


Figure 6-5: An Example of A Test Run

As the test runs, a pop-up window appears indicating the test instructions' status of PASS/FAIL as shown in **Figure 6-6** below.

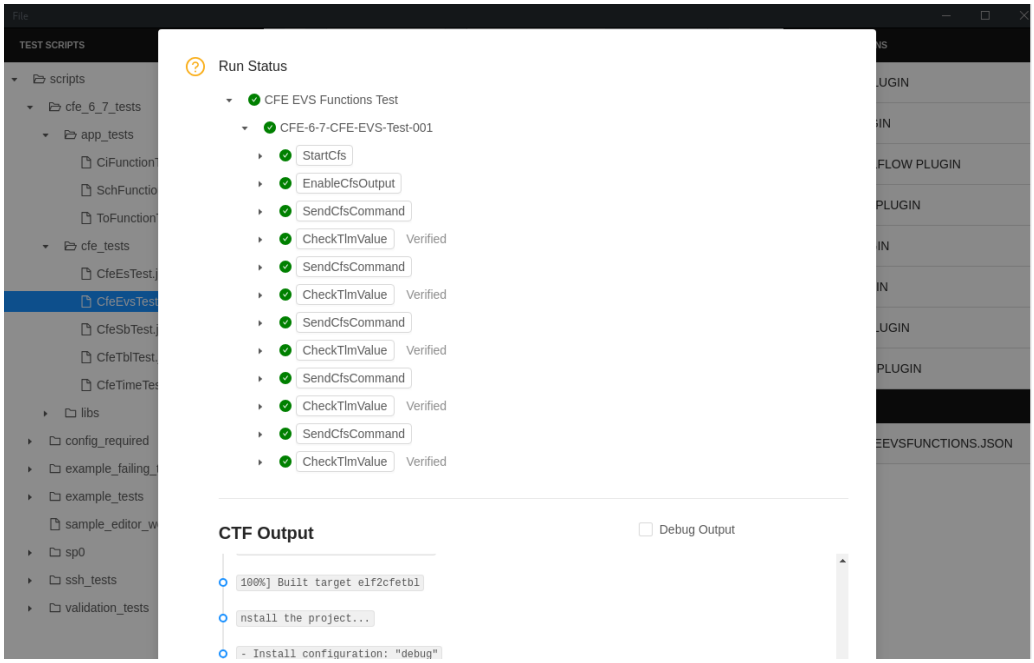


Figure 6-6: An Example of a Test Run with Execution Status

Verify that this is the correct version before use

6.4 How to Create a New CTF Plugin

CTF plugins are implementations of the various CTF test instructions. The plugins allow the users to extend CTF with new or custom test instructions. Each test instruction is mapped to its respective implementation in the plugin.

To create a new CTF plugin, create a directory with the **<plugin-name>** under the **plugins** directory. For example, the users can create a new plugin called **example_plugin** at the directory, **ctf/plugins/example_plugin**.

Not that plugins must conform to a naming convention to be loaded by CTF. The module containing the plugin class must end with **_plugin** and not contain the word "**tests**" in its module path. This is because the unit tests for the plugins are placed under **<plugin-name>/tests** directory. For more information, see the CTF built-in plugins as examples.

In the **plugins/example_plugin** directory, create a new Python file called **example_plugin.py** that contains the plugin class implementation, which is required to inherit from the CTF built-in **Plugin** base class. Each plugin is required to define the following properties:

- **name** - A plugin name that is unique across all CTF plugins.
- **description** - A description of the plugin.
- **command_map** - A Python dict object mapping between each test instruction name to a Python function name that implement that test instruction.
- **verify_required_commands** - A Python list of test instruction names that require verification.

The **example_plugin.py** will look as follows:

```
from lib.plugin_manager import Plugin, ArgTypes
from lib.logger import logger as log

class ExamplePlugin(Plugin):
    # ------------------
    # Constructor method
    # ------------------
    def __init__(self):
        self.name = "ExamplePlugin"
        self.description = "CTF Example Plugin"
        self.command_map = {
            "TestCommand": (self.test_command, [ArgTypes.string] * 2),
            "TestVerifyCommand": (self.test_verify_command, [])
        }
        self.verify_required_commands = ["TestVerifyCommand"]
        self.example_counter = 0
    # ---------------------
    # Initialization method
    # ---------------------
    def initialize(self):
        #print("Initialize runs before a script is executed")
        return True
    # --------------------------------------
    # Method for TestCommand test instruction
```

```
    # ----------------------------------------
    def test_command(self, arg1, arg2):
        log.info("Test Command Executed with args: {}, {}".format(arg1, arg2))

        # Command implementation goes here...

        # Return status of that instruction
        status = True
        return status
    # ----------------------------------------------
    # Method for TestVerifyCommand test instruction
    # ----------------------------------------------
    def test_verify_command(self):
        log.info("Test Verify Executed")

        # Non-blocking verification code goes here...

        # Here, we intentionally wait for example_counter > 5
        # before allowing the verification to pass

        self.example_counter += 1
        if self.example_counter > 5:
            status = True
        else:
            status = False

        # Return status of verification
        return status
    # -----------------
    # Destructor method
    # -----------------
    def shutdown(self):
        log.info("Optional shutdown/cleanup implementation for plugin")
```

With the above plugin definition, the following snippet of a JSON test script shows how the new test instructions are used.

```
{
    "test_number": "Example-Plugin-Test",
    "test_name": "Example Plugin Test",
    "requirements": {
        "REQT-01": "The system shall …"
    },
    "description": "Testing Example Plugin",
    "owner": "CTF",
    "test_setup": "Script executes a single instruction, and a single verification",
    "ctf_options": { "verify_timeout": 2.0 },
    "tests": [
        {
            "case_number": "Example Plugin Test",
            "description": "No description",
            "instructions": [
                {
                    "instructions": "TestCommand",
                    "wait": 1,
```

```
                    "data": {
                        "arg1": "foo",
                        "arg2": 42
                    }
                },
                {
                    "instructions": "TestVerifyCommand",
                    "wait": 1,
                    "data": {}
                }
            ]
        }
    ]
}
```

7 ANATOMY OF A CTF TEST SCRIPT

A test script is defined in JSON format. (For more detailed information on JSON, see https://www.w3schools.com/js/js_json_syntax.asp.) A test script is comprised of a few properties that describe the tests. Each test script has one or more test cases. Each test case has one or more test instructions and/or test functions. Example test scripts can be found in the **`functional_tests`** directory of the CTF repository.

7.1 Test Script Definition

A test script has the following properties:

- **test_number**: A unique identifier for the test script.
- **test_name**: A short descriptive name for the test script.
- **requirements**: A list of requirement IDs that the test script verifies.
- **description**: A description of the test scenario or any notes associated with the test script.
- **import**: A list of test functions being imported from other JSON files.
- **functions**: One or more user-defined local test functions used by the test script.
- **tests**: A list of test cases, where each test case contains one or more test instructions and/or test functions.

Below is an example.

```
{
    "test_number": "Basic-Cfs-Plugin-Test",
    "test_name": "Basic CFS Plugin Test",
    "requirements": {
        "CFS-1": "The system shall …",
        "CFS-20": "The system shall …",
    },
    "description": "Basic CTF Example Script Showing Simple Commands/Telemetry Verification",
    "owner": "CTF",
    "test_setup": "Script will start ctf, execute a verification instruction and close ctf",
    "ctf_options": { "verify_timeout": 4 },
    "import": {},
    "tests": [
        {
            "case_number": " Cfs-Plugin-Test-001",
            "description": "Start CFS, Send TO NOOP instruction",
            "instructions": [
```

```
                {
                    "instruction": "StartCfs",
                    "wait": 1,
                    "data": {}
                },
                {
                    "instruction": "CheckTlmValue",
                    "wait": 1,
                    "data": {
                        "mid": "TO_HK_TLM_MID",
                        "args": [
                            {
                                "compare": "==",
                                "variable": "usCmdErrCnt",
                                "value": [ 0 ]
                            }
                        ]
                    }
                },
                {
                    "instruction": "SendCfsCommand",
                    "wait": 1,
                    "data": {
                        "mid": "TO_CMD_MID",
                        "cc": "TO_NOOP_CC",
                        "subsysId": 7,
                        "endian": 0,
                        "systemId": 0,
                        "args": []
                    }
                },
                {
                    "instruction": "CheckTlmValue",
                    "wait": 1,
                    "data": {
                        "mid": "TO_HK_TLM_MID",
                        "args": [
                            {
                                "compare": "==",
                                "variable": "usCmdCnt",
                                "value": [ 2 ]
                            }
                        ]
                    }
                },
                {
                    "instruction": "ShutdownCfs",
                    "wait": 1,
                    "data": {}
                }
            ]
        }
    ]
}
```

7.2 Test Function Definition

A test function is a group of test instructions that can be called by a test case. These functions can be defined within the same test script or in a separate file and be imported.

A test function has the following properties:

- **description**: A description of the purpose of the function including the steps the function with take and the results the function will provide. The details on the intended use should also be included.
- **varlist**: A list of function arguments.
- **instructions:** A list of test instructions to execute**.**

Below is an example.

```
"functions":{
    "SendCheckToResetCmd":{
     "description": "Send and check TO_RESET_CC",
     "varlist": [ "expectedCmdCnt"],
     "instructions": [
         {
             "instruction": "SendCfsCommand",
             "wait": 0,
             "data": {
                 "target": "",
                 "mid": "TO_CMD_MID",
                 "cc": "TO_RESET_CC",
                 "args": {}
             }
         },
         {
             "instruction": "CheckTlmValue",
             "wait": 0,
             "data": {
                 "target": "",
                 "mid": "TO_HK_TLM_MID",
                 "args": [
                     {
                         "variable": "usCmdCnt",
                         "value": [ "expectedCmdCnt"],
                         "compare": "=="
                     }
                 ]
             }
         }
     ]
}
```

7.3 Test Case Definition

A test case describes a single test. It has the following properties:

- **case_number**: A unique identifier of the test case in the script.
- **description**: A description of what actions this test is performing, including the preconditions, the expected actions/instructions, and the expected results.
- **instructions**: A list of test instructions and/or test functions that define the actions taken by the test.

Below is an example.

```
{
    "case_number":"ExampleTest-1",
    "description":"<Explain preconditions, actions, results, steps of test case>",
    "instructions": [
        {
```

```
            "instruction": "StartCfs",
            "wait": 1,
            "data": {}
        },
        {
            "function": "SendCheckToResetCmd",
            "wait": 0,
            "data": {}
        }
    ]
}
```

7.4 Test Instruction Definition

A test instruction defines a single action. There are 2 types of supported test instructions:

a. Non-verification test instructions

   These instructions do not require verification at a later time and can be validated right away. For example, sending a cFS command is a non-verification test instruction.

b. Verification test instructions

   These instructions require verification (via polling) until the verification is satisfied, or a timeout is reached. For example, checking that a piece of telemetry changes at some point in the test is a verification test instruction. Note that the of verification instructions must be non-blocking in order to return control to CTF between the verification polls.

Each instruction object has the following properties:

- **instruction:** The action to be performed. The action is implemented via CTF plugins.
- **data:** The list of arguments for the instruction.
- **wait:** The amount of time to wait before the instruction is executed.
- **timeout:** (Optional) The amount of time to wait for the verification to succeed before timing out.

Instructions are implemented in their respective CTF plugins. Refer to the specific plugin's README document for more information on instructions supported by plugins.

The test script examples can be found at the `functional_tests` folder of the CTF repository.

Table 7.1 lists the available test instructions.

Table 7-1: CTF Test Instruction Reference

8 ASSUMPTIONS, DEPENDENCIES AND CONSTRAINTS

8.1 Assumptions

None.

8.2 Dependencies

The CTF tool uses the 3rd-party software packages as listed in **Table 4-1** above.

8.3 Constraints

None.

9 LIMITATIONS AND WARNINGS

9.1 Limitations

None.

9.2 Warnings

None.

10 KNOWN PROBLEMS

The CTF's known problems and known changes are documented in the CTF's Version Description Document (VDD).

11 APPENDICES

11.1 Frequently Asked Questions

11.1.1 General CTF

a. What platforms does CTF support?

CTF is developed and tested on CentOS 7 Linux. It may work with other Linux distributions or Docker containers, but is not officially supported.

b. What version of CTF am I running?

Check the first line in the terminal output or of **`CTF_Log_File.log`** in your log output directory, for

```
*** INFO: cFS Test Framework (vX.X) Starting...
```

where X.X is the version number. For the latest version, see https://github.com/nasa/CTF/releases.

c. Can I use environment variables in paths? How do I refer to files outside of the CTF workspace?

CTF will expand system environment variables and the **`~`** symbol in paths using standard Linux syntax. You can use either absolute paths or paths relative to the location of CTF. Within the config file you can also reference config values in other entries using the syntax "**`${section:key}`**" as seen in the default config file.

To assist in importing files in test scripts, **`workspace_dir`** from the **`[cfs]`** section of the config may be used as an environment variable.

d. I can't run the new CTF release. I'm getting error relating to missing applications/packages.

Whenever you first run a new CTF release, it is highly recommended that you remove your existing anaconda3 directory, and source the script, **`setup_ctf_env.sh`**, again to pick up any new system packages that got added since the last release.

11.1.2 CTF Configurations

a. How do I configure a CFS target using SSH? What config keys do I need to include for my CFS target?

Targets using the SSH protocols require additional keys in their config sections. See the included config files for more information and examples.

For SSH targets (where "**`cfs_protocol = ssh`**"):

- **`destination`**

All keys must be specified in the config file, however they may appear in either the named target section or the default **`[cfs]`** section. Keys in the named target section will override values in the **`[cfs]`** section. In other words, **`[cfs]`** may provide fallback values for any keys not found in named target sections, including SSH targets.

11.1.3 cFS Startup and Shutdown

a. I see an error "**`pidof: command not found`**" in CTF output, and/or cFS does not shut down?

The **`pidof`** command is used to stop cFS processes on Linux. You may need to install the **`sysvinit-tools`** package in your environment to enable **`pidof`**.

b. I have unexpected or missing telemetry, or **`EnableCfsOutput`** is failing.

A cFS instance may still be running in the background and holding on to resources. Try

i. Execute

```
$ ps -aux | grep core
```

(or the name of your cFS exe) before and/or after a test run to check for any persisting processes.

ii. Rebooting the host computer may also help if some other process is using a required port.

iii. Check the **`cmd_udp_port`** and **`tlm_udp_port`** configuration items in the CTF INI file, in the **`[cfs]`** section, to make sure they match with those used by the cFS instance. The command port is where CTF sends commands to cFS, which should match CI app's port. The telemetry port is where cFS send telemetry to CTF, which should match TO app's port.

b. How do I run multiple test scripts at once?

When executing CTF test scripts, you may provide any number of directory and/or file paths to be executed in order. For directories, all JSON files will be executed in determinate order.

Example:

```
$ ./ctf --config_file=./configs/default_config.ini test_scripts script1.json
script2.json script3.json
```

This will execute all JSON files in the **`test_scripts`** directory, followed by **`script1`**, **`script2`**, and finally **`script3`**.

If the tests are designed to be executed in sequence, set **`reset_plugins_between_scripts`** to False in your CTF INI config file to allow the plugin state, and potentially the cFS instances, to be preserved across all test scripts - similarly to multiple test cases in one script. This allows you to avoid registering and starting cFS targets in each test script. Note that when doing so, cFS will not be stopped until the end of the final test, unless **`ShutdownCfs`** is used.

c. When should I use **`RegisterCfs`**, or use a target name versus an empty string?

Using **`RegisterCfs`** to declare your cFS target(s) is optional. The CTF Plugin will automatically register and use CFS targets as defined in the config file if no target name is explicitly provided. Automatic registration occurs at the first occurrence of **`BuildCfs`** or **`StartCfs`** if no target has been registered. During automatic registration, any section in the config file beginning with **`cfs_`** will be treated as a cFS target, such as **`[cfs_lx1]`**. The required section **`[cfs]`** will be used to register a default

target if no others are provided. When explicitly registering a target with `RegisterCfs`, the name must match a section in the config file, but it is not required to begin with cfs_. In other words, `[cfs_lx1]` may be registered automatically or with `RegisterCfs` but `[lx1]` can only be registered with `RegisterCfs`.

Like test script, you may or may not provide a target name for any CFS Plugin instruction. If a target name is provided, it must match a section name in the config file, whether it was registered explicitly or automatically. If no target is provided (an empty string or omitting `target` altogether), the instruction will apply to all registered targets sequentially. Thus, omission of target names allows a test script to be trivially reused with different config files, but is most practical when configured with a single target.

These features allow you to mix and match test scripts with different configurations by relying on automatic registration of targets. If the test script does not name or register any targets, its instructions will apply to any targets as defined in the config file. Alternatively, you may use `RegisterCfs` to specify only those targets from your config file that you want to register for the test, even if there are others defined.

d. How should I register a target if I need to perform SSH commands directly? Why can't I connect to CFS after registering a SSH target?

Registering a CFS target with `RegisterCfs` takes precedence over `SSH_RegisterTarget`. If you are interacting with CFS use `RegisterCfs` regardless of the protocol. You can then interact with the target by name using SSH commands, as applicable. Only use `SSH_RegisterTarget` if you are only using SSH plugin features, but not interacting with CFS.

11.1.4 Test Script Execution

a. Why is CFS stopping between test scripts, or a CFS target not valid in subsequent scripts? How do I keep CFS alive between test scripts?

The config key `reset_plugins_between_scripts` (boolean) tells CTF whether to automatically shut down and recreate plugins between each test script (but not between test cases in the same script), which among other things will stop CFS and clear any registered targets.

When running multiple test scripts for a single process instance, set `reset_plugins_between_scripts` to False.

To shut down CFS during a test, or between tests when `reset_plugins_between_tests` is disabled, use `StopCfs`.

11.1.5 CTF Test Script Syntax

a. I'm getting unusual Python errors relating to my instruction arguments?

Instructions that evaluate expressions, like `CheckTlmValue`, support different operators for comparing values. Standard logical operators like `==, !=, <=`, etc are allowed for numeric comparisons. When comparing strings, use `streq` for equality or `strneq` for inequality. You cannot use string comparison for numbers or vice versa.

Types used in JSON should match the corresponding data as closely as possible. Use whole numbers for integers, decimals for floats or doubles, and strings for strings or characters. Do not put a numeric value in quotes unless it is to be handled as a string.

For complex types, such as "`args`" for `SendCfsCommand`, use the nested JSON objects with key-value pairs to represent the structure. Within an object, you may use an ordered list for values that are arrays (i.e., those having a nonzero `array_size` attribute in their data definition). Do not use lists to represent multiple fields in an object.

To specify a single value in an array, you may use the indexed name of the element (e.g., `"myArray[3]" = 1`). You may also provide a single value to an array name to fill the array with that value (e.g., `"myArray" = 1` sets ALL elements in `myArray` to 1). Both methods can be used in combination. For examples of correct usage of these and other instruction arguments, see the test scripts in `scripts/example_tests`.

b. Why am I seeing config validation errors after updating to CTF v1.4/v1.5?

CTF v1.4 / v1.5 introduces a few new config keys for Linux targets:

- `cfs_ram_drive_path` (string, Linux only): a path, if provided, to clear persistent memory when starting CFS without the "`-RPR`" argument in `cfs_run_args`

You can use the CTF upgrade script, `tools/upgrade_v1_5.py`, to add these keys and default values to config files.

c. Why do event checks look different in CTF v1.4 / v1.5? Why are my `CheckEvent` and `CheckNoEvent` instructions producing parameter errors after updating to CTF v1.4 / v1.5?

The syntax for `CheckEvent` and `CheckNoEvent` changed in CTF v1.4 / v1.5 to resemble that of `CheckTlmValue`. This allows scripts to check for multiple events in the same instruction, which can be useful when several events are expected in an unknown order. See `plugins/cfs/README.md` for details on instruction syntax and usage. You can use the CTF upgrade script, `tools/upgrade_v1_5.py`, to update the syntax of existing test scripts.

It is important to note than when checking for multiple events, that all event checks must pass in the same polling attempt. This means that each of the checked events must be received within the window between when EVS messages are cleared before polling, `evs_messages_clear_after_time`, and the instruction timeout, `ctf_verification_timeout`. If messages are expected to arrive at significantly different times it is probably best to use separate instructions.

11.1.6 cFS Events and Telemetry Verifications

a. `CheckEvent` or `CheckTlmValue` is failing, but I see the events in CTF logs.

Your checks may be too early or too late. For event checks, CTF will only check events that arrive between the start of the check and the verification timeout. Compare the execution times in `cfs_tlm_msgs.log` with the timestamps of the checks in the CTF test output file. (Note that execution

time is the offset in seconds from the start of the test.) We recommend using a "`wait = 0`" for all checks, unless you know that the packet will be delayed:

For example,

```
{
      "instruction": "EnableCfsOutput",
      "wait": 2,
      "data": {}
},
{
      "instruction": "CheckEvent",
      "wait": 0,
      "data": {
          "app": "TO",
          "id": "3",
          "msg": "TO - ENABLE_OUTPUT cmd successful for routeMask:0x00000001"
      }
},
...
```

b. My first telemetry check for a given packet passes, but others fail.

Checking any value in a packet clears that packet from memory, to avoid checking stale data again later in the test. If you need to check multiple values from the same packet, combine the conditions in one instruction using a list:

For example,

```
"args": [
    {
         "compare": "==",
         "variable": "cmdCount",
         "value": 1
    },
    {
        "compare": "==",
        "variable": "errCount",
        "value": 0
    },
...
```

11.1.7 Logging

a. How do I get color-coded log output?

Simply install the Python package `colorlog: (pip install colorlog)`. The package is not installed by default since many users run CTF via the editor or do not inspect the output at runtime. Colors only apply to the command line output.

b. How do I capture and view logs from the target?

CTF creates several log files in a timestamped directory under `results_output_dir` for each test run. These include

- the full CTF console output broken down by test case
- the full cFS execution stdout and build stdout if applicable
- all received cFS telemetry packets and EVS messages
- a summary of the test run in both JSON and CSV format

You can change the logging level of CTF itself with `log_level`.

For SSH targets only, the config key, `print_stdout`, tells CTF to print command output inline in console output.

For SSH targets, the config key, `log_stdout`, (boolean) tells CTF to log the stdout response to each command on completion.

Depending on the test these settings may be very verbose, so it is recommended to use them for debugging only and to refer to log files for the full output.

11.1.8 Miscellaneous

a. I am seeing some other odd behavior not described here.

Make sure you are using the latest CTF and cFS. Pull from the Git repositories, check out the latest release branch/tag, re-install your Anaconda environment, and rebuild everything. Try running CTF directly without the editor or intermediate scripts. If the problem persists, contact our CTF development team or open an issue at https://github.com/nasa/CTF/issues.

b. Where can I find more information on the NASA-only plugins, like the SP0 plugin?

If this document is from the NASA-only release of CTF, i.e., not a https://github.com/nasa/ctf CTF release, the information is in CTF_SUG_FAQ-SP0.pdf.

11.2 Abbreviations and Acronyms

Term	Definition
API	Application Programming Interface
BSP	Board Support Package
CCDD	CFS Command and Data Dictionary Tool
CCSDS	Consultative Committee for Space Data Systems
cFE	Core Flight Executive
cFS	Core Flight System
CI	Command Ingest cFS Application
COTS	Commercial Off-the-Shelf
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
EA	JSC Engineering Directorate Organization Code
ES	cFE Executive Services
EVS	cFE Event Services
GFE	Government Furnished Equipment
HK	Housekeeping cFS Application
JSC	NASA Johnson Space Center
MDT	Message Definition Table (for SCH_TT application)
MID	Message Identifier
NASA	National Aeronautics and Space Administration
NPR	NASA Procedural Requirements
OS	Operating System
OSAL	cFS Operating System Abstraction Layer
PSP	cFS Platform Support Package
SB	cFE Software Bus services
SBNg	Software Bus Network for Gateway cFS Application
SCH	Scheduler cFS Application

Term	Definition
SCH_TT	Time-Triggered Ethernet Scheduler cFS Application
SDD	Software Detailed Design
SDT	Schedule Definition Table (for SCH_TT application)
SRS	Software Requirements Specification
TBD	To Be Determined
TDM	Time-Division Multiplexer
TO	Telemetry Output (cFS application)
TTE	Time-Triggered Ethernet
TTE ES	Time-Triggered Ethernet End System
TTE_LIB	Time-Triggered Ethernet cFS Library

11.3 Definition of Terms

None.

12 NOTES

None.