

NASA Operational Simulator for Small Satellites



User Manual & Developer's Guide



Independent Verification and Validation (IV&V)

NASA IV&V
100 University Dr,
Fairmont, WV 26554
304.367.8200

Point of Contact
NOS³ Support
support@nos3.org



Table of Contents

1	Introduction	5
1.1	Background	5
1.2	Format	6
2	NOS³ Architecture	7
3	NOS Engine	8
4	Ground Systems	9
4.1	AMMOS Instrument Toolkit (AIT)	9
4.2	COSMOS	10
4.3	Command Telemetry Link Up	10
5	Quick Start to Installing, Building, and Running NOS³	12
5.1	Installing NOS ³	12
5.2	Building NOS ³	12
5.3	Running NOS ³	13
5.4	Running COSMOS	14
6	Detailed Installation and Virtual Machine Creation Steps	16
6.1	The Vagrantfile and Process	17
6.1.1	Vagrant Plugins	17
6.1.2	Base Virtual Machine Configuration	17
6.1.3	Provisioning the Virtual Machine	18
6.1.4	Conclusion and References	18
7	Building NOS³ Components: Flight Software and Simulators	20
7.1	Detailed Build Steps	20
7.2	The Makefile	22
8	Running NOS³: Standalone Server, Simulators, 42, Flight Software, and COSMOS	24
9	NOS³ Workflows	30
10	Hardware Simulator Framework / Example Simulator	32
10.1	Background and Supporting Concepts	32
10.1.1	Abstract Factory Design Pattern	32
10.1.2	XML Configuration	32
10.2	Implementing Your Own Hardware Model (and Data Provider, and Connections)	33
10.2.1	Configuration Data Property Tree	33
10.2.2	Hardware Model	34
10.2.3	Data Provider	34
10.2.4	Connections	35
10.3	Writing Your Own Simulator	37
10.4	Example Simulator	39
11	42, A Visualization and Simulation Tool for Spacecraft Orbit and Attitude Dynamics	40
11.1	42 Overview	40
11.2	Providing Data to a Simulator from 42	41
11.3	Coordinating 42 Time	41
11.4	Data Available from 42	42
12	Flight Software Development, Especially Using cFS	43
12.1	cFS and NOS ³	43
12.1.1	Operating System Abstraction Layer	43
12.1.2	Platform Support Package	43
12.1.3	Hardware Library	43
12.2	Connecting cFS to NOS ³	44
12.3	NOS ³ Drivers and Other FSW	45

12.3.1	Writing a NOS ³ Driver	45
13	Hardware In The Loop	48
14	Orbit, Inview, and Power Planning Tool.....	49
15	FAQ	50
1	Introduction	4
1.1	Background.....	4
1.2	Format.....	5
2	NOS³ Architecture.....	6
3	NOS Engine.....	7
4	Ground Systems.....	8
4.1	AMMOS Instrument Toolkit (AIT).....	8
4.2	COSMOS.....	9
4.3	Command Telemetry Link Up.....	9
5	Quick Start to Installing, Building, and Running NOS³.....	11
5.1	Installing NOS ³	11
5.2	Building NOS ³	11
5.3	Running NOS ³	12
5.4	Running COSMOS.....	13
6	Detailed Installation and Virtual Machine Creation Steps	15
6.1	The Vagrantfile and Process	16
6.1.1	Vagrant Plugins	16
6.1.2	Base Virtual Machine Configuration	16
6.1.3	Provisioning the Virtual Machine	17
6.1.4	Conclusion and References	18
7	Building NOS³ Components: Flight Software and Simulators.....	19
7.1	Detailed Build Steps	19
7.2	The Makefile	21
8	Running NOS³: Standalone Server, Simulators, 42, Flight Software, and COSMOS	23
9	NOS³ Workflows	29
10	Hardware Simulator Framework / Example Simulator	31
10.1	Background and Supporting Concepts.....	31
10.1.1	Abstract Factory Design Pattern	31
10.1.2	XML Configuration	31
10.2	Implementing Your Own Hardware Model (and Data Provider, and Connections).....	32
10.2.1	Configuration Data Property Tree	32
10.2.2	Hardware Model.....	33
10.2.3	Data Provider.....	33
10.2.4	Connections.....	34
10.3	Writing Your Own Simulator	36
10.4	Example Simulator.....	38
11	42, A Visualization and Simulation Tool for Spacecraft Orbit and Attitude Dynamics.....	39
11.1	42 Overview.....	39
11.2	Providing Data to a Simulator from 42.....	40
11.3	Coordinating 42 Time.....	40
11.4	Data Available from 42.....	41
12	Flight Software Development, Especially Using cFS.....	42
12.1	cFS and NOS ³	42
12.1.1	Operating System Abstraction Layer.....	42
12.1.2	Platform Support Package	42
12.1.3	Hardware Library.....	42



<u>12.2</u>	<u>Connecting cFS to NOS³</u>	<u>43</u>
<u>12.3</u>	<u>NOS³ Drivers and Other FSW</u>	<u>44</u>
<u>12.3.1</u>	<u>Writing a NOS³ Driver</u>	<u>44</u>
<u>13</u>	<u>Hardware In The Loop</u>	<u>47</u>
<u>14</u>	<u>Orbit, Inview, and Power Planning Tool</u>	<u>48</u>
<u>15</u>	<u>FAQ</u>	<u>49</u>



Table of Figures

Figure 1 - NOS3 Architecture	<u>76</u>
Figure 2 - AIT Display	<u>98</u>
Figure 3 - AIT in Action.....	<u>109</u>
Figure 4 – COSMOS Sender Tool.....	<u>1140</u>
Figure 5 - COSMOS cFS communication	<u>1140</u>
Figure 6 - NOS3 Virtual Machine Complete	<u>1244</u>
Figure 7 - Building Simulators and Flight Software	<u>Error! Bookmark not defined.12</u>
Figure 8 - NOS3 Running - Server, Sumulators, 42, Flight Software, COSMOS	<u>Error! Bookmark not defined.14</u>
Figure 9- COSMOS Command Sending - Enabling Telemetry	<u>Error! Bookmark not defined.14</u>
Figure 10 - COSMOS Command and Telemetry Server - Showing "Bytes Tx", "Bytes Rx"	<u>15</u>
Figure 11 - COSMOS Command and Telemetry Server - Selecting "View in Packet Viewer" for "NOS3_NAV_MSG"	<u>1645</u>
Figure 12 - COSMOS Packet Viewer - NOS3_NAV_MSG	<u>Error! Bookmark not defined.16</u>
Figure 13 - Initial Desktop.....	<u>Error! Bookmark not defined.18</u>
Figure 14 - Ubuntu Linux Desktop Greeter	<u>2024</u>
Figure 15 - Double click "first-nos3-build.sh"	<u>Error! Bookmark not defined.22</u>
Figure 16 - NOS3 Building: CMake Execution	<u>2123</u>
Figure 17 - NOS3 Building: Make and Install the Flight Software	<u>2123</u>
Figure 18 - NOS3 Building: Make and Install Simulators.....	<u>2224</u>
Figure 19 - Ubuntu Linux Desktop Greeter	<u>2426</u>
Figure 20 - Double Click "nos-3-run.sh"	<u>2527</u>
Figure 21 - NOS Engine Standalone Server.....	<u>2528</u>
Figure 22 - 42 Dynamic Simulator.....	<u>2629</u>
Figure 23 - COSMOS	<u>2730</u>
Figure 24 - Simulators	<u>2834</u>
Figure 25 - NOS3 Flight Software	<u>2932</u>
Figure 26 - Shared Folder Settings.....	<u>3033</u>
Figure 27 - Share in NOS3 Folder	<u>3134</u>
Figure 28 - Flight and Simulation Targets.....	<u>4447</u>
Figure 29 - Example OIPP Report	<u>4953</u>



1 Introduction

This document, titled “NOS³ User Manual and Developer’s Guide”, provides information for users and developers that intend to enhance and extend the NASA Operational Simulator for Small Satellites (NOS³).

1.1 Background

The NASA Independent Verification and Validation (IV&V) Independent Test Capability (ITC) team developed a 3U Small Satellite named Simulation-to-Flight 1 (STF-1). The primary goal of this Small Satellite was to develop and demonstrate the lifecycle value of a software-only small satellite simulator. This simulator is called the NASA Operational Simulator for Small Satellites or NOS³.

NOS³ is an open-source, software only testbed for small satellites licensed under the NASA Open Source Agreement. It is a collection of Linux executables and libraries. Current simulations are based on commercial-off-the-shelf (COTS) hardware that is being used on the STF-1 CubeSat. It is intended to easily interface with flight software developed using the NASA Core Flight System (cFS).

NOS³ executes on an Ubuntu Linux virtual machine and is comprised of a number of components. These components are listed in the following table.

Component	Description
Vagrant	Vagrant is an open source solution that can be used to script the creation of Oracle VirtualBox virtual machines and the provisioning of such machines, including package installation, user creation, file and directory manipulation, etc.
VirtualBox	Oracle VirtualBox is an open source solution for creating and running virtual machines.
NOS Engine	NASA Operational Simulator (NOS) Engine is a NASA developed solution for simulating hardware busses as software only busses. This component provides the connectivity between the flight software and the simulated hardware components.
Simulated Hardware Components	A collection of simulated hardware components which connect to NOS Engine and provide hardware input and output to the flight software.
42	Some of the hardware components require dynamic environmental data. 42 is an open source visualization and simulation tool for spacecraft attitude and orbital dynamics developed by NASA Goddard Space Flight Center (GSFC) which is used to provide dynamic environmental data.
cFS	NASA Core Flight Software (cFS) is an open source Flight Software used as the base system which STF-1 flight software is developed on.
AIT	Is a light weight open source ground system developed by JPL that provides command and control to the flight software.
COSMOS	COSMOS is open source ground system software developed by Ball Aerospace which is used to provide command and control of the flight software.
OIPP	Orbit , Inview, and Power Planning (OIPP) is an ITC developed planning tool which can use current two line element (TLE) sets from the internet or a TLE file to project satellite to ground station inview times and satellite eclipse and sunlight times.



1.2 Format

The format of this document is as follows. Section 2 describes the overall architecture of NOS³, including the component architecture and how the components communicate with each other.

Section 3 describes NOS Engine and how it is used by developers to provide the software bus interface between flight software and simulated hardware.

Section 4 describes the AIT and COSMOS ground systems and how they can be used to interact with the sample telemetry output (TO_Lab) application and sample command ingest (CI_Lab) application that are provided with cFS.

Section 5 is a quick start guide with a minimum set of procedures for creating the NOS³ virtual machine, building the flight software and simulator components on the NOS³ virtual machine, and running the NOS engine standalone server, simulators, 42, flight software, and COSMOS in order to have end to end command and control, flight software execution, and simulation.

Section 6 provides detailed instructions for creating the NOS³ virtual machine and describes the steps that occur in the Vagrantfile to configure and provision the virtual machine.

Section 7 provides detailed instructions for building the flight software and simulations on the NOS³ virtual machine.

Section 8 provides detailed instructions for running the NOS engine standalone server, simulators, 42, flight software, and COSMOS and describes the various components that are automatically started using the quick start script.

Section 9 elaborates on the various types of NOS³ workflows that exist. This includes setting NOS³ for editing and using version control on your host or inside of the VM.

Section 10 describes the framework for developers to use to develop hardware simulators and provides information on example simulator code included with NOS³.

Section 11 describes the 42 visualization and simulation tool for spacecraft attitude and orbital dynamics which is used to provide environmental data to those simulators that need it to provide realistic data.

Section 12 describes developing flight software using cFS and interfacing it with NOS engine and the simulators.

Section 13 explains the hardware in the loop capabilities while expanding on the installation and use for each platform.

Section 14 describes the Orbit, Inview, and Power Planning (OIPP) tool. This tool is not part of the end to end command and control simulation suite of NOS³ that can be used during flight software development, but provides a planning tool for use in preparing for, testing, and executing mission operations.



2 NOS³ Architecture

Figure 1 shows the architecture of NOS³. To get started with NOS³, a NOS³ user needs to install Oracle VirtualBox and Vagrant on their host computer. Both of these software packages are open source and can be run on various operating systems, including Microsoft Windows, Apple OS X, and Linux. In addition to those software packages, NOS³ is comprised of a collection of files that are stored in a git repository. To get started with NOS³, the user receives a copy of those files and places them on their computer. These files include a Vagrantfile, which is a file that is used by the Vagrant software package to create an Ubuntu Linux Virtual Machine where all of NOS³ is run. During creation of the Ubuntu Linux Virtual Machine, various software packages will be installed via Ansible, including AIT, COSMOS, 42, and the NOS Engine libraries and NOS Standalone Server. An alternative to starting with Vagrant is to receive an already generated VirtualBox Virtual Machine with the various packages installed. To build and run the core flight software, simulators, and so on, the source code will need to be present as described below.

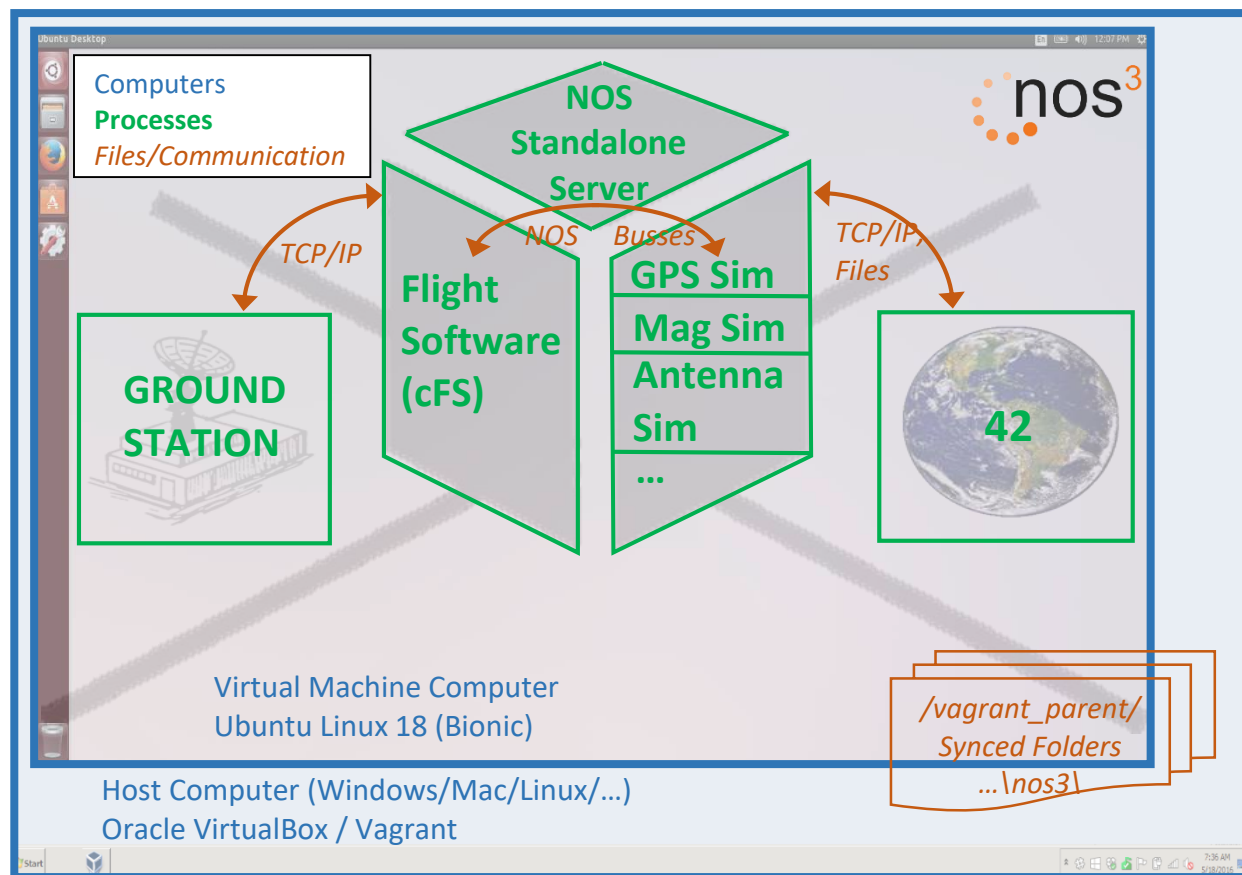


Figure 1 - NOS³ Architecture

Source code for various simulators is accessible on the virtual machine through shared folders which allow access to the same files on the host computer and the virtual machine computer. Build tools can be used on the virtual machine to build and install simulators such as a GPS simulator, a magnetometer simulator, an antenna simulator, and more. In addition, two special software tools are built and installed as part of the simulators. The first is a NOS time driver that provides time ticks to drive time for the various simulators, 42,



and the flight software. The second is a simple terminal program which can be used by the operator to command and control other simulators using the NOS engine command bus on which all of the simulators can be nodes.

The cFS source code is also present on the virtual machine through the shared folders. Build tools can also be used to build and install the generic flight software. This flight software includes hardware libraries that can interface as nodes on NOS Engine busses in place of the real hardware node and bus connections.

As shown in Figure 1, TCP/IP or files can be used to provide environmental data from 42 to the various simulators. In addition, TCP/IP can be used to interface COSMOS with laboratory versions of command and telemetry applications in cFS. Finally, the NOS Engine libraries are used to provide the software busses and nodes for communication between the flight software and the simulated hardware and for distribution of simulation time.

3 NOS Engine

NOS Engine is a message passing middleware designed specifically for use in simulation. With a modular design, the library provides a powerful core layer that can be extended to simulate specific communication protocols, including I2C, SPI, and CAN Bus. With advanced features like time synchronization, data manipulation, and fault injection, NOS Engine provides a fast, flexible, and reusable system for connecting and testing the pieces of a simulation.

NOS Engine is built on a conceptual model based on two fundamental types of objects: nodes and buses. A node is any type of endpoint in the system capable of sending and/or receiving messages. Any node in the system has to belong to a group, referred to as a bus. A bus can have an arbitrary number of nodes, and each node on the bus must have a name that is unique from other member nodes. The nodes of a bus operate in a sandbox; a node can communicate with another node on the same bus, but cannot talk to nodes that are members of a different bus.

Within NOS³, NOS Engine is used to provide software simulations of hardware buses. NOS Engine provides the infrastructure for each hardware simulator to be a node on the appropriate bus and for the flight software to interact with hardware simulator nodes on their bus. NOS Engine also provides plug-ins for various protocols such as MIL-STD-1553, SpaceWire, I2C, SPI, CAN, and UART. These plug-ins allow each bus and the nodes on that bus to communicate using calls and concepts specific to that protocol.

For more information on the concepts, architecture, specific bus protocols supported, or other information on using NOS Engine, please refer to the NOS Engine User's Manual.

4 Ground Systems

NOS3 supports two ground systems out of the box - AIT and COSMOS.

4.1 AMMOS Instrument Toolkit (AIT)

AIT is a Python-based software suite developed by JPL to handle Ground Data System (GDS), Electronic Ground Support Equipment (EGSE), commanding, telemetry uplink/downlink, and sequencing for JPL International Space Station and CubeSat Missions.

If AIT is the ground station utilized, the project repository is installed under the AIT directory and a python virtual environment is created during installation. The default AIT installation is under the "ait" virtual environment and a cFS specialized project is installed under the "ait-cfs" virtual environment. If the user wants to create their own Ground Station using AIT they may use the ait virtual environment. This document uses the AIT-cFS installed in the "ait-cfs" virtual environment. AIT web interface is pictured below.

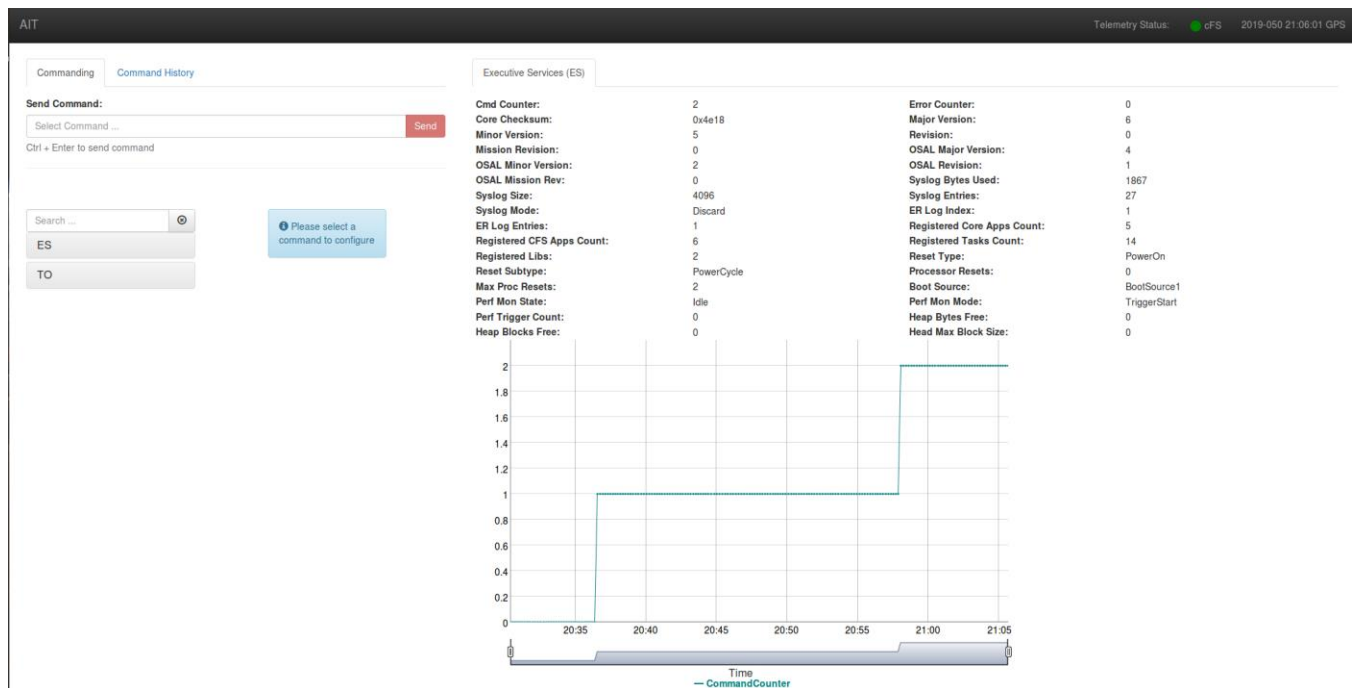


Figure 2 - AIT Display



4.2 COSMOS

COSMOS is an open source ground system provided via Ball Aerospace¹ and is included with NOS³ to provide an alternate ground station for the simulated spacecraft. COSMOS is installed in its default directory and run from nos3/gsw/cosmos.

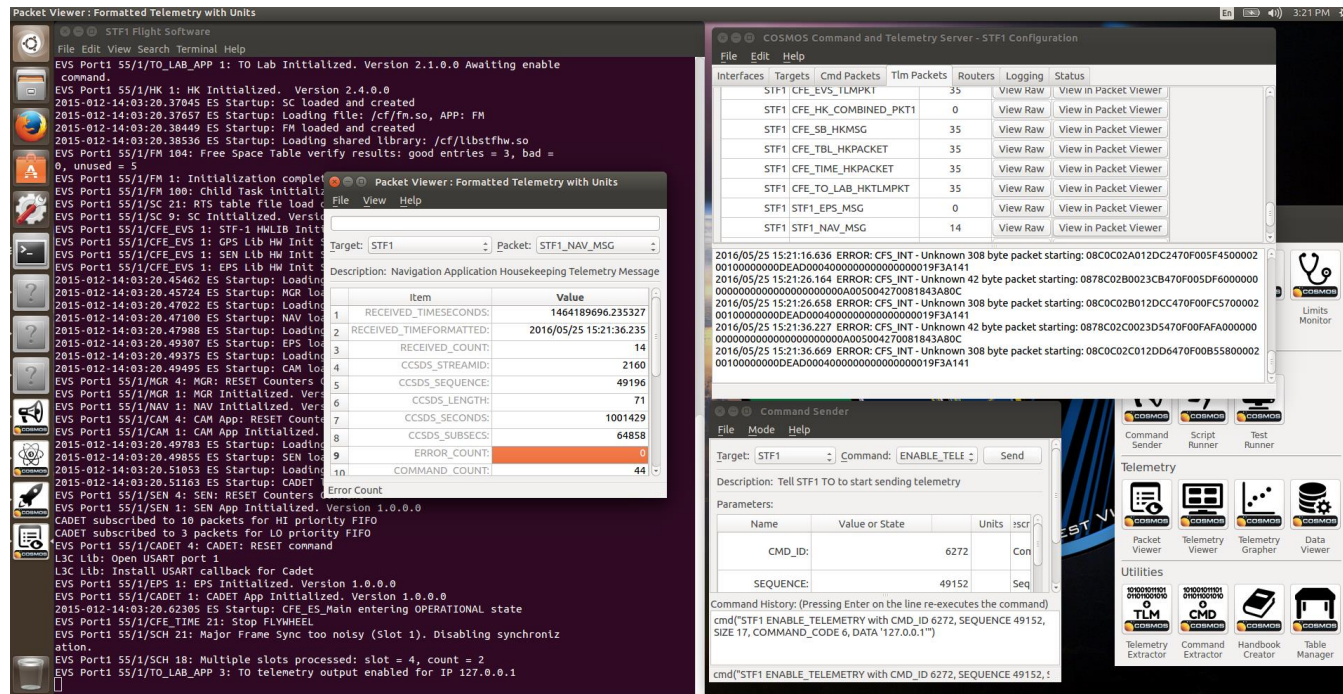


Figure 3 - AIT in Action

4.3 Command Telemetry Link Up

The link to a Ground Station is completed by two applications in cFS. These are the command ingest (CI) and telemetry output (TO) applications. In NOS³ these apps utilize UDP to communicate and are not meant for flight operations. The TO link is closed by default on start-up, but can be activated by sending a specific command packet. Commanding is done by using the Command Sender tool in COSMOS using the target named 'CFS' with a single command to 'TO_ENABLE_OUTPUT_CC'. Once sent, the TO app will reply stating that telemetry is enabled. This is demonstrated in the screenshot below. It should be noted that only telemetry listed in the 'to_config.h' will be captured. Additional telemetry can be appended as necessary.

¹ <http://cosmosrb.com/>

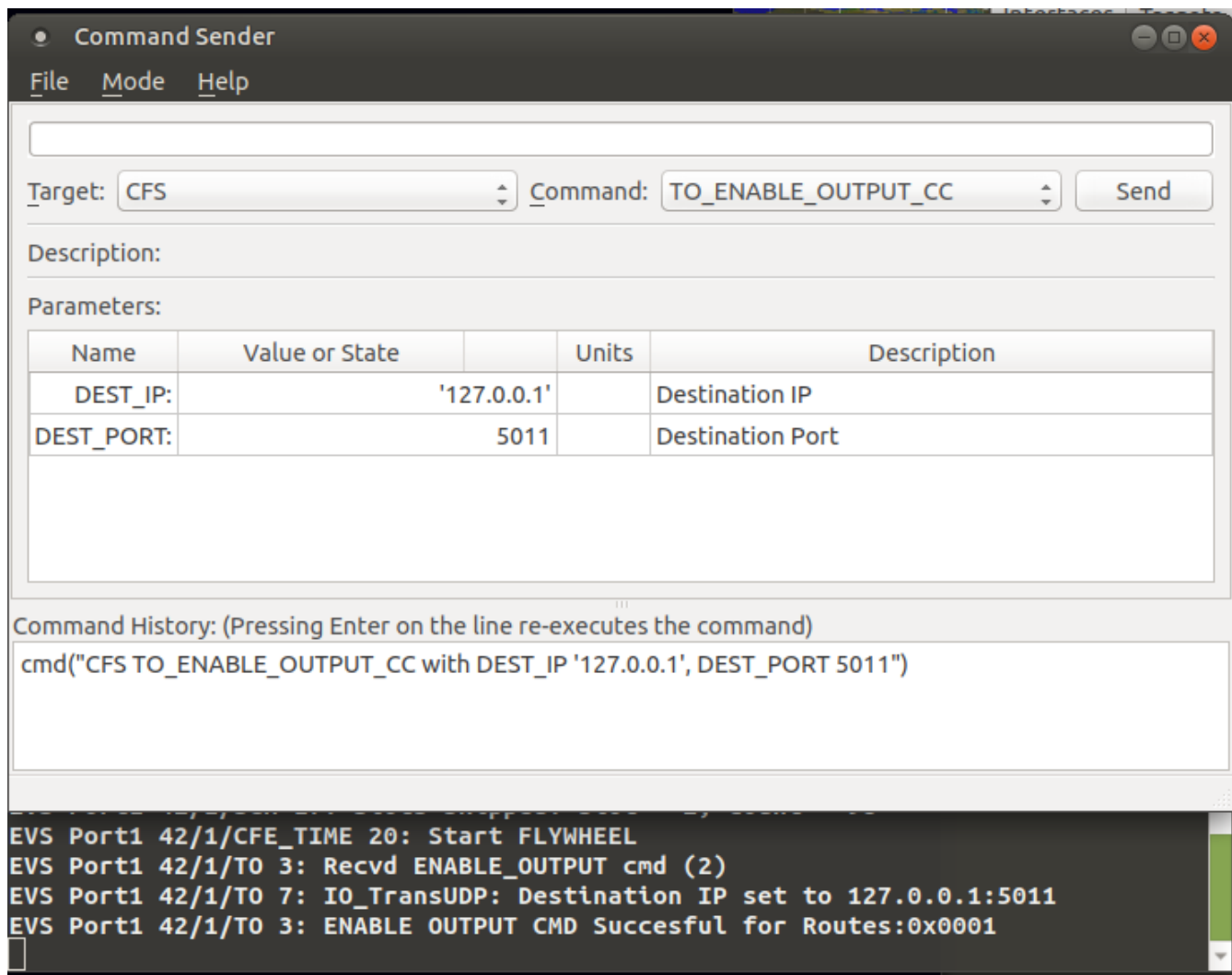


Figure 4 - COSMOS Sender Tool

All communications to, from, and internal to cFS are formatted using the CCSDS standard packet type with the secondary header enabled. This secondary header allows the specific command to be passed to the application specified in the primary header. COSMOS requires knowledge of these commands and telemetry structures to be able to construct and interpret them as needed. An example is provided below:

```
COMMAND EPS EPS_NOOP BIG_ENDIAN "EPS NOOP"
  APPEND_PARAMETER CCSDS_STREAMID 16 UINT MIN_UINT16 MAX_UINT16 0x18C0 "CCSDS Packet
  Identification" BIG_ENDIAN
  APPEND_PARAMETER CCSDS_SEQUENCE 16 UINT MIN_UINT16 MAX_UINT16 0xC000 "CCSDS Packet Sequence
  Control" BIG_ENDIAN
  APPEND_PARAMETER CCSDS_LENGTH 16 UINT MIN_UINT16 MAX_UINT16 1 "CCSDS Packet Data Length"
  BIG_ENDIAN
  APPEND_PARAMETER CCSDS_CHECKSUM 8 UINT MIN_UINT8 MAX_UINT8 0 "CCSDS Command Checksum"
  APPEND_PARAMETER CCSDS_FC 8 UINT MIN_UINT8 MAX_UINT8 0 "CCSDS Command Function Code"
```





Figure 5 - COSMOS cFS communication



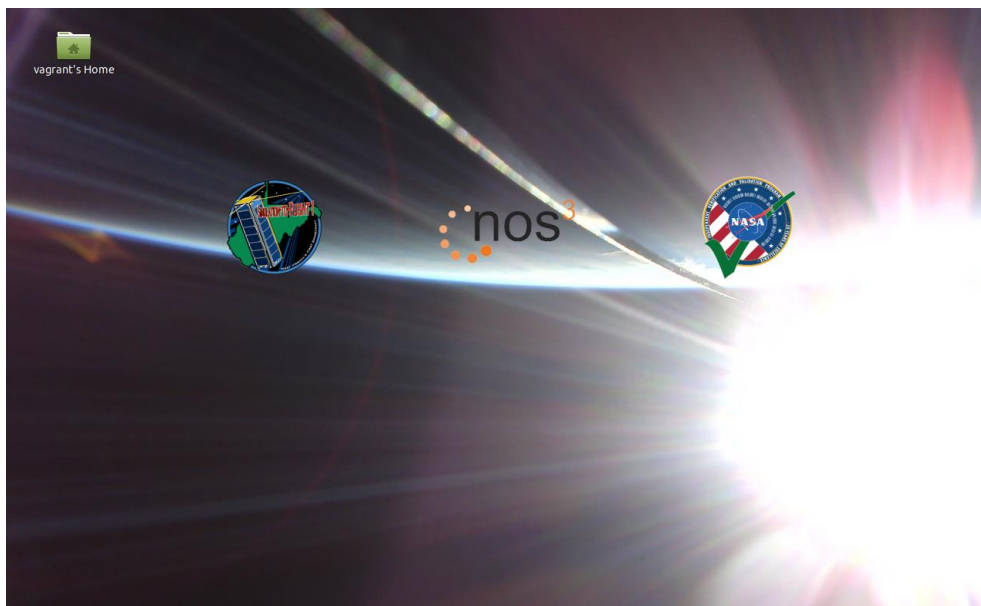
5 Quick Start to Installing, Building, and Running NOS³

5.1 Installing NOS³

On the host computer:

1. Install Oracle VirtualBox v6.1 + (<https://www.virtualbox.org/>) 
2. Install Vagrant v2.2+ (<https://www.vagrantup.com/>) 
3. Install Git 1.8 + (<https://git-scm.com/downloads/>) 
4. Acquire the *nos3* release repository via git clone. 
5. Initialize git submodules for use:
 - a. git submodule init
6. Update git submodules:
 - a. git submodule update
7. Navigate to *nos3/deployment*
8. Configure the CONFIG file in the deployment directory:
 - a. In this file you can choose the configuration settings for the VM that will be created by Vagrant:
 - i. Operating Systems are CentOS(1) and Ubuntu(2) **CentOS CURRENTLY NOT SUPPORTED**
 - ii. Ground Systems are AIT(1) and COSMOS(2)
9. Run the command *vagrant up* via a command prompt within the *nos3/deployment* directory, and wait for the command to return to the prompt. – This can take anywhere from 20 minutes to hours depending on internet speeds and the specs of the host PC.
10. Vagrant will automatically load the virtual machine to Virtual Box, and it will be ready for use.

Login to the *nos3* user using the password *nos3123!*
Login to the *vagrant* user using the password *vagrant*



5.2 Building NOS³

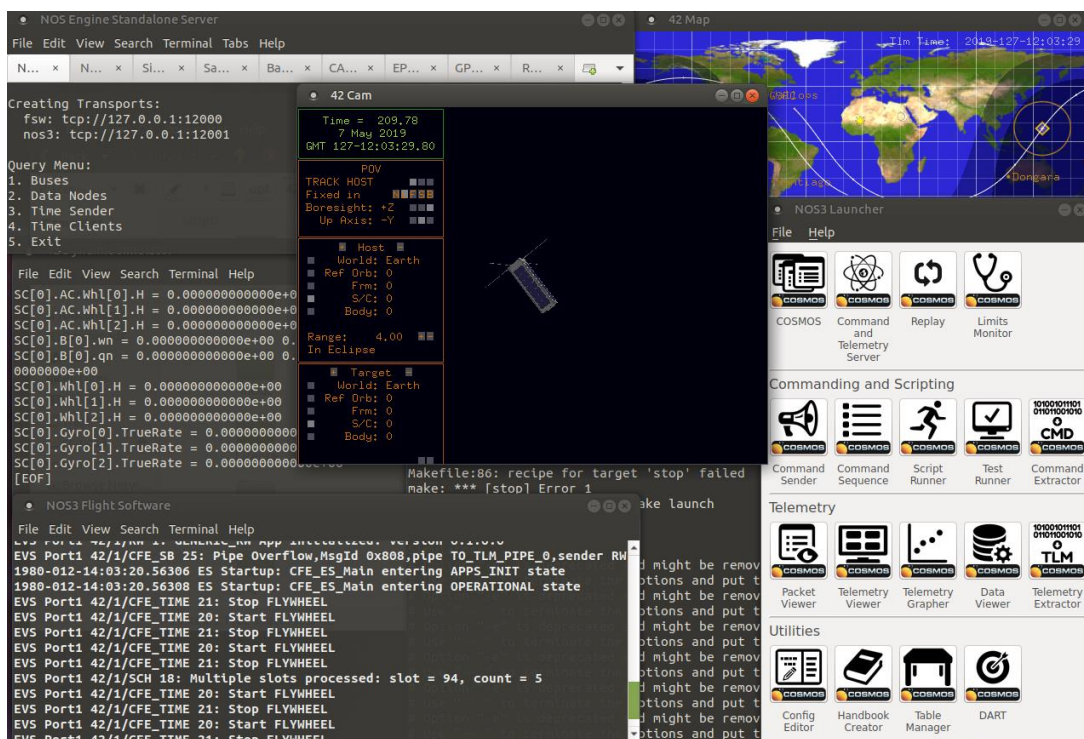
Log in to the NOS³ VM, username / password : nos3 / nos3123!

- 1) Access the nos3 repository on the Host Machine. From the Guest:
 - a) Use Virtual Box – Guest Additions CD Add
 - b) Virtual box Menu > Devices > Shared Folder > Shared Folder Settings
 - c) Add new Shared Folder and select the location of nos3 repo on host
 - d) Check Select Auto Mount, Make Permanent
 - e) Reboot VM

- 2) Make and Install software
 - a) Navigate to the nos3 shared folder after VM Reboot from Step 1e
 - b) Navigate to the nos3 shared folder and nos3 parent directory
 - c) Run “make”

5.3 Running NOS³

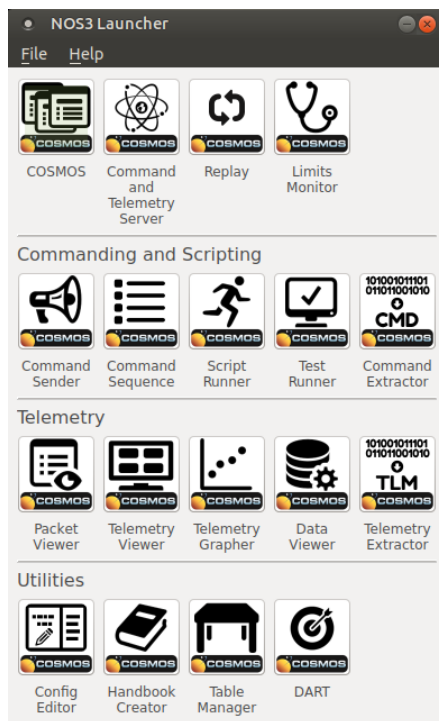
- 1) Navigate to the nos3 shared folder and nos3 parent directory
- 2) Run “make launch” in the nos3 parent directory from the terminal



- 3) To exit the simulation run “make stop” from the terminal
- 4) To rebuild NOS³ from the repository baseline, first run “make clean” then “make”.

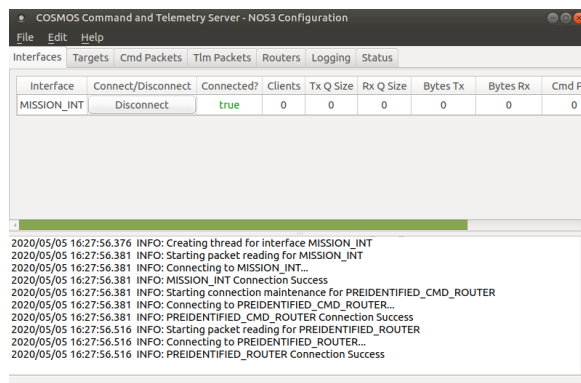
5.4 Running COSMOS

Once NOS³ is launched, the cFS flight software and COSMOS ground station will start.



COSMOS may command cFS to send telemetry back to COSMOS by:

1. Open the *Command and Telemetry Server* and Enter:
 - i. `-- config nos3_cmd_tlm_server.txt`
 - ii. `- system nos3_system.txt`
 - iii. The Connected? attribute ought to read "True"





2. Open COSMOS *Command Sender*, under “Commanding and Scripting,” and Enter:
 - i. In the Commander Sender Options Window – system ***nos_system.txt***
 - ii. In the Command Sender Window, Select *Target:* to be “***CFS***”
 - iii. Under *Command:* select ***TO_ENABLE_OUTPUT_CC***,
 - iv. Click “***Send***”

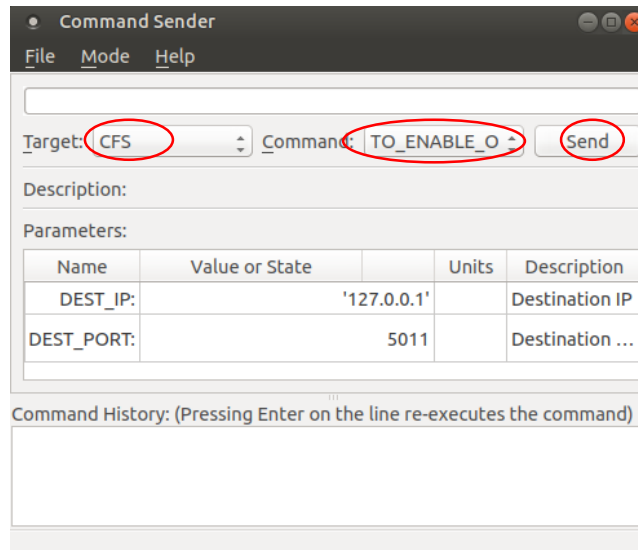


Figure 69 - COSMOS Command Sending - Enabling Telemetry

- v. Notice under COSMOS Command And Telemetry Server Window, the data field updates
 1. “*Bytes Tx*” and “*Cmd Pkts*” should change from 0 to a positive number
 2. “*Bytes Rx*” and “*Tlm Pkts*” should start counting up as telemetry is received

Interface	Connect/Disconnect	Connected?	Clients	Tx Q Size	Rx Q Size	Bytes Tx	Bytes Rx	Cmd Pkts	Tlm Pkts	View Raw
MISSION_INT	Disconnect	true	0	0	0	26	1188	1	8	View Raw

Figure 710 - COSMOS Command and Telemetry Server - Showing “Bytes Tx”, “Bytes Rx”

3. Open COSMOS “*Packet Viewer*”
 - i. – system ***nos3_system.txt***
 - ii. Select *Target:* to be “***CFS***” Select Packet Name “***CFE_EVS_PACKET***”
 - iii. Scroll to see the “*Message*” field live updates (Line 16)
 - iv. This may be repeated for other apps, once they are commanded to send telemetry
 - v. Fields of applications which have not received telemetry packets are in **fuchsia**

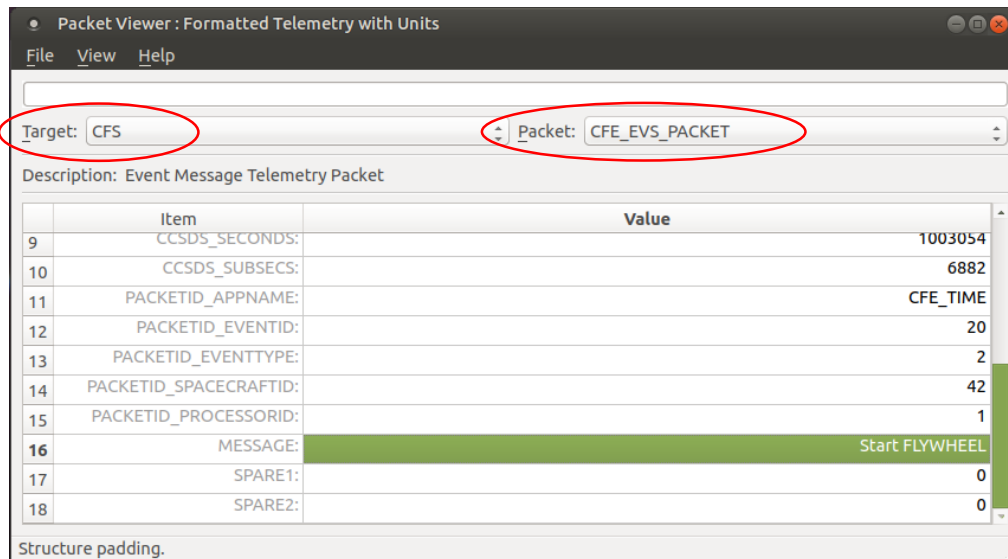


Figure 811 - COSMOS Packet Viewer

6 Detailed Installation and Virtual Machine Creation Steps

As mentioned in the background and quick start sections, the key prerequisite to being able to install and run NOS³ on a user's computer is the installation of Oracle VirtualBox and Vagrant. Information and installers for these products can be found at:

1. Oracle VirtualBox – <https://www.virtualbox.org/>
2. Vagrant - <https://www.vagrantup.com/>

Following installation of these products, the next prerequisite for installing and running NOS³ is to obtain the nos3 code repository. Currently, that repository has a folder structure like the following:

- ~/nos3/ contains the repository at the time of the build locally in the VM.
 - /fsw
 - /apps - the open source cFS apps
 - /build - the unarchived build directory
 - /cfe - the core flight system (cFS) source files
 - /components – cFS app support for flight hardware
 - /nos3_defs – common definitions for cFS-based missions when building in nos3
 - /osal - operating system abstraction layer (OSAL), enables building for multiple flight OS
 - /psp - platform support package (PSP), enables use on multiple types of boards
 - /tools - standard cFS provided tools
 - gsw - all the files needed for ground stations and installation
 - /ait - AIT configuration files
 - /cosmos - COSMOS database files
 - /OrbitInviewPowerPrediction - OIPP tool for operators
 - /scripts – build and installation scripts
 - deployment



- /ansible - installation scripts
- /nos3_filestore
 - /backgrounds – VM wallpaper
 - /packages - installation packages
- Vagrantfile - main provisioner file used to generate the VM
- /sims – source code for simulated hardware components
- .gitignore - list of files and directories to be omitted from git
- .gitmodules - list of git submodules utilized
- Makefile - top level makefile to be used for the build directory
- README.md – Basic information contained within this Guide and about the nos3 repository

IMPORTANT: Internet access is **required** when installing. Also, please do **NOT** log in to the virtual machine until the provisioning process is complete and vagrant has finished. All Figures captured were produced from a Windows install.

6.1 The Vagrantfile and Process

The following section describes the provisioning that is done in the NOS³ Vagrantfile using Vagrant. Vagrantfiles are text files written in a language called Ruby.

6.1.1 Vagrant Plugins

The first items in the Vagrantfile configure optional Vagrant plugins that may or may not be installed in the user's environment and which can make virtual machine provisioning easier. These really only benefit the user if multiple vagrant runs take place. The first is a plugin called vagrant-vbguest, which attempts to keep the VirtualBox guest additions software up-to-date if newer versions of VirtualBox are installed on the user's machine. The second is a plugin called vagrant-cachier. It attempts to cache packages that are downloaded from the internet as part of virtual machine configuration and provisioning. Once the packages are cached, the time consuming process of re-downloading them from the internet can be avoided. The final plugin, vagrant-reload, aids in the provisioning process and provides a means to reboot the machine without losing the current position in the installation.

6.1.2 Base Virtual Machine Configuration

The next item defines the base box or base virtual machine configuration which is used for the virtual machine. In the case of NOS³, this base box is a very minimal installation of Ubuntu Linux, Version 18 (Bionic). This minimal installation is mainly intended as a server installation with no graphical desktop.

When Vagrant starts the NOS3 virtual machine, it automatically creates a synced folder between the host and the virtual machine. In the host, that synced folder is the directory containing the Vagrantfile (*nos3\support*). On the Linux VM, this folder appears as the directory */vagrant*. In addition, the Vagrantfile specifies the creation of an additional synced folder between the host and the virtual machine so that the other source code and files that are part of the nos3 folder are available on the virtual machine. In the host, that synced folder is *../*, or one level up from where the Vagrantfile exists (*nos3*). On the Linux VM, this folder appears as the directory */vagrant_parent*.



Finally, the Vagrantfile contains some initial configuration information for the virtual machine, including the name to give the VM, the fact that the GUI should be displayed, the amount of memory and number of CPUs to give the VM, the ability to have a DVD drive, and several parameters controlling the graphics capabilities to assign to the VM.

This concludes the basic configuration of Vagrant and the virtual machine.

6.1.3 Provisioning the Virtual Machine

The next section in the Vagrantfile consists of a shell provisioner. This shell provisioner is a series of Linux shell commands that are run by the root user in the VM in order to configure it to have the packages, users, directories, and other configuration settings needed for NOS³. The shell provisioner section consists of the following subsections.

First, there is a section containing additional package installation commands to install Python, Linux headers, build tools, debuggers, utilities, GUI toolkits, a minimal desktop environment, web browsers, and other needed packages. This is followed by the installation of AIT or COSMOS.

After these tools have been installed, the next set of commands installs NOS Engine, additional common functionality provided by the Independent Test Capability (ITC) team, and the 42 open source visualization and simulation tool for spacecraft attitude and orbital dynamics.

After that, several configuration settings are altered to increase the number of message queues, to set the path for finding dynamic libraries, and to keep core dumps locally rather than sending them to the Ubuntu community.

The next section adjusts the user accounts on the virtual machine. It deletes the Ubuntu user if present, disables the guest user, and adds the nos3 user.

After that, several preferences are changed for the backgrounds and so that double clicking executable scripts runs them instead of viewing them in an editor.

Next, the nos3 user's desktop environment is configured by copying various scripts, symbolically linking several directories to appear in convenient locations, and installing and configuring COSMOS for the nos3 user.

Then several Python packages are installed and several scripts copied to the nos3 user's environment that support mission planning.

Finally, VirtualBox Guest Additions are installed/updated for the desktop environment if the desktop environment is running.

6.1.4 Conclusion and References



This concludes an overview of the Vagrantfile which is used to install and configure the NOS³ environment. For more details, please consult the Vagrantfile itself.

7 Building NOS³ Components: Flight Software and Simulators

7.1 Detailed Build Steps

To elaborate on the quick start guide, once the NOS³ virtual machine is created, the steps to build the flight software and simulators are:

1. Use “vagrant up” from the nos3/support directory or start from the Oracle VM VirtualBox Manager
2. Once the Ubuntu Linux virtual machine desktop greeter appears, log in using:
 - a. Username: nos3
 - b. Password: nos3123!

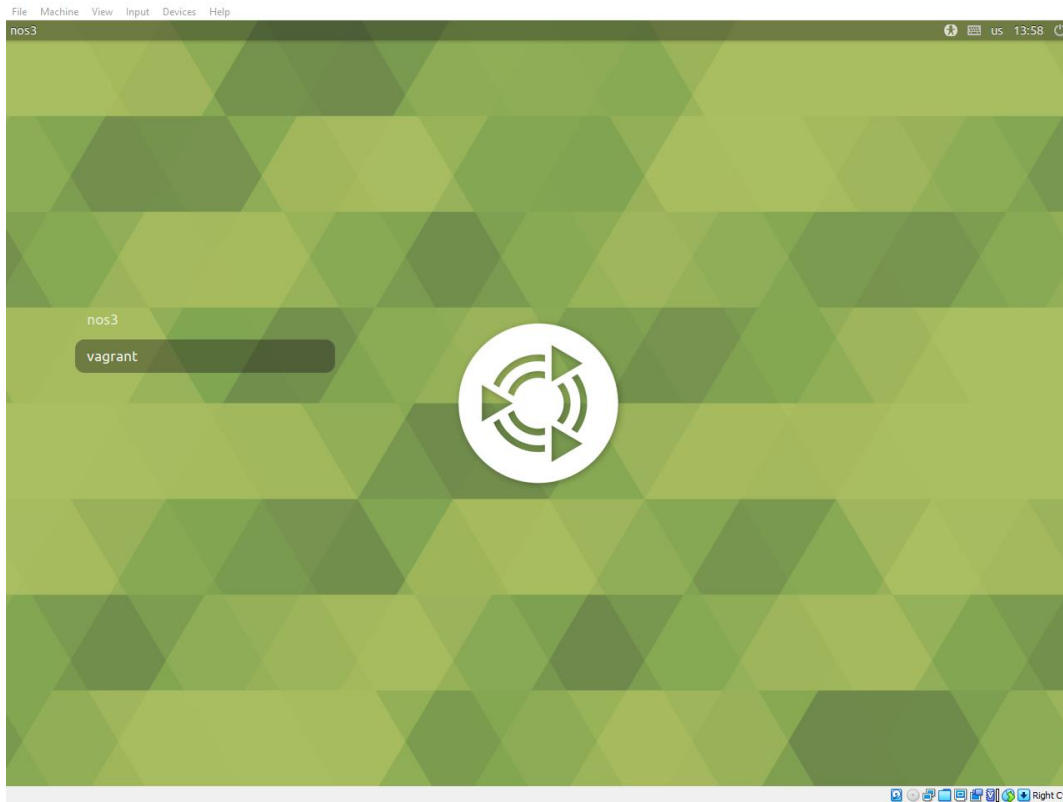


Figure 914 - Ubuntu Linux Desktop Greeter

3. Create a shared folder with the host at the host's /nos3 git repository
4. Navigate to the parent directory of the nos3 shared folder and run “make”



```

-DBUILD_SIMULATOR=YES -G "CodeBlocks - Unix Makefi
-- Module 'hk' found at /home/nos3/nos3/apps/hk
-- Module 'eps' found at /home/nos3/nos3/stf_apps/eps
-- Module 'libstfhw' found at /home/nos3/nos3/stf_apps/libstfhw
-- Module 'nav' found at /home/nos3/nos3/stf_apps/nav
-- Module 'cam' found at /home/nos3/nos3/stf_apps/cam
-- Module 'sample_stf1_app' found at /home/nos3/nos3/stf_apps/sample_stf1_app
-- Module 'sc' found at /home/nos3/nos3/apps/sc
-- Module 'cfs_lib' found at /home/nos3/nos3/apps/cfs_lib
-- Module 'ds' found at /home/nos3/nos3/apps/ds
-- Module 'fm' found at /home/nos3/nos3/apps/fm
-- Module 'hs' found at /home/nos3/nos3/apps/hs
-- Module 'sen' found at /home/nos3/nos3/stf_apps/sen
-- Module 'cfe-core' found at /home/nos3/nos3/cfe/fsw/cfe-core
-- Configuring for system arch: linux
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/g++
-- Check for working CXX compiler: /usr/bin/g++ -- works
-- Detecting CXX compiler ABI info

```

Figure 1016 - NOS3 Building: CMake Execution

```

vagrant_parent -DBUILD_SIMULATOR=YES -G "CodeBlock
make mission-all # CFE apps, etc
1 warning generated.
[ 30%] Building C object cfe/cfe_core_default_linux/CMakeFiles/cfe_core_default_
linux.dir/src/time/cfe_time_tone.c.o
[ 30%] Building C object cfe/cfe_core_default_linux/CMakeFiles/cfe_core_default_
linux.dir/src/time/cfe_time_utils.c.o
[ 31%] Building C object cfe/cfe_core_default_linux/CMakeFiles/cfe_core_default_
linux.dir/src/fs/cfe_fs_api.c.o
[ 32%] Building C object cfe/cfe_core_default_linux/CMakeFiles/cfe_core_default_
linux.dir/src/fs/cfe_fs_decompress.c.o
/vagrant_parent/cfe/fsw/cfe-core/src/fs/cfe_fs_decompress.c:703:18: warning:
comparison of unsigned expression >= 0 is always true
[-Wtautological-compare]
        if ( index >= 0 && index < State->hufts )
            ^
/vagrant_parent/cfe/fsw/cfe-core/src/fs/cfe_fs_decompress.c:750:27: warning:
comparison of unsigned expression >= 0 is always true
[-Wtautological-compare]
        if ( index >= 0 && index < State->hufts )
            ^
2 warnings generated.
[ 33%] Building C object cfe/cfe_core_default_linux/CMakeFiles/cfe_core_default_
linux.dir/src/fs/cfe_fs_priv.c.o
Linking C static library libcfe_core_default_linux.a

```

Figure 1117 - NOS3 Building: Make and Install the Flight Software



```
vagrant_parent -DBUILD_SIMULATOR=YES -G "CodeBloc
make mission-all # CFE apps, etc
[ 90%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/src/bcr.cp
P.o
[ 91%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/src/pcm.cp
P.o
[ 91%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/src/pdm.cp
P.o
[ 91%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/src/eps.cp
P.o
[ 91%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/test/versi
on_test.cpp.o
[ 92%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/test/statu
s_test.cpp.o
[ 92%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/test/bus_t
est.cpp.o
[ 92%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/test/comma
nd_test.cpp.o
[ 93%] Building CXX object sim/eps_sim/libeps/CMakeFiles/test_eps.dir/test/main.
cpp.o
Linking CXX executable test_eps
[ 93%] Built target test_eps
Scanning dependencies of target eps_sim
[ 93%] Building CXX object sim/eps_sim/eps_sim/CMakeFiles/eps_sim.dir/src/config
.cpp.o
```

Figure 1218 - NOS3 Building: Make and Install Simulators

- When this process is complete, a new folder “*build*” should exist in the fsw and sim repos with the built software.

7.2 The Makefile

Inside the makefile, the cmake scripts for subsequent flight software and simulation components are being prepped and invoked:

- cmake*

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.

In the VM, the source for the flight software and the simulators is located in the directory “~/nos3”. When Make runs, it preps and procs cmake files in child directories to create makefiles and other build files necessary for compiling, linking, and installing the flight software and simulators. These files are created in the shared folder directories “nos3/fsw/build” and “nos3/sim/build”.

- make* and *make launch*

Make gets its knowledge of how to build your program from a file called the *makefile*, which lists each of the non-source files and how to compute it from other files. When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program.



When “*make*” is run, *cmake* files are invoked to build the flight software, sims, and ground system which are compiled and linked in “*build*” subdirectories of their respective directory. When “*make launch*” is run, NOS3 launches the flight software, 42 simulator, NOS Engine, and ground station. This includes the “*core-linux*” executable of *cFS* and “*cf*” directory which contains the shared objects, libraries, and configuration tables and files for the flight software.

8 Running NOS³: Standalone Server, Simulators, 42, Flight Software, and COSMOS

To elaborate on the quick start guide, once the NOS³ virtual machine is created and the flight software and simulators are built, all of the software comprising NOS³ can be run:

1. Use “vagrant up” from the nos3/support directory or start from the Oracle VM VirtualBox Manager
2. Once the Ubuntu Linux virtual machine desktop greeter appears, log in using:
 - a. Username: nos3
 - b. Password: nos3123!

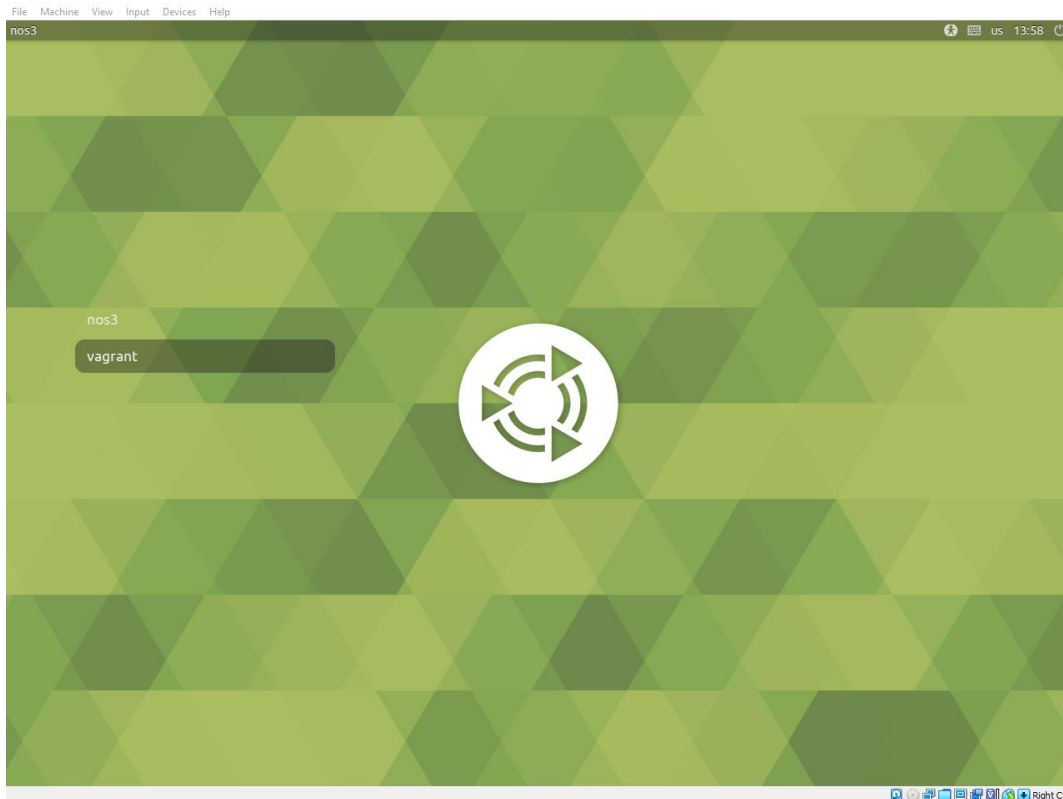


Figure 1319 - Ubuntu Linux Desktop Greeter

3. Create the NOS³ git repository shared folder “/nos3” and run “make”.

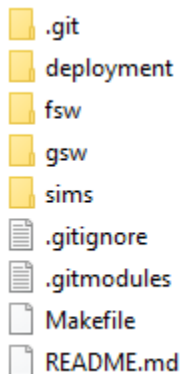


Figure 1420 - Open a Terminal in the /nos3 parent directory; run “make” then “make launch”

4. The following software will start up:
 - a. NOS Engine Standalone Server (1 terminal window)

```

NOS Engine Standalone Server
File Edit View Search Terminal Help
ry document entity '/home/stf1/Desktop/stf1-build/bin/nos_engine_log_config.xml'
  at line 0 col 0
Configuration File Name: /home/stf1/Desktop/stf1-build/bin/nos_engine_server_stf
1_simulator_config.json

Loading Plugins: uart

Creating Transports:
  fsw: tcp://127.0.0.1:12000
  nos3: tcp://127.0.0.1:12001

* STANDALONE SERVER APPLICATION *

- This application may need to be killed manually using CTRL+C

Query Menu:
1.   Buses
2.   Data Nodes
3.   Time Clients
4.   Time Sender
5.   Exit

Selection:
    
```

Figure 1521 - NOS Engine Standalone Server

The NOS Engine Standalone Server provides the software simulated communication bus structure that is used by NOS³ to connect the flight software with simulated flight hardware. NOS Engine Standalone Server is installed when the ITC NOS Engine package is installed. The executable is *nos_engine_server_standalone*. For NOS³, the server is configured using the file *nos3/sims/build/bin/nos_engine_server_config.json* which defines plugin protocols and uniform resource identifiers (URIs) for the server.

- b. 42 Dynamic Simulator (1 terminal window, 1 GUI window with CubeSat, 1 GUI window with map)

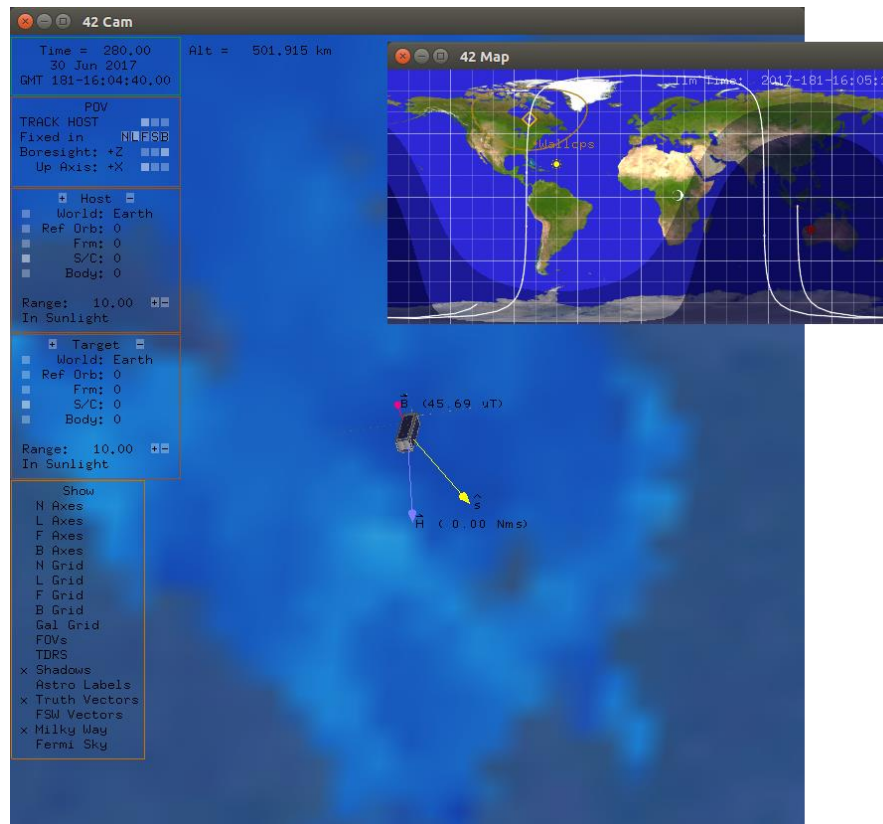


Figure 1622 - 42 Dynamic Simulator

42 is a general-purpose, multi-body, multi-spacecraft simulation. For NOS³, it simulates the motion of the STF-1 cubesat. The progression of time for 42 is driven through NOS Engine and 42 provides output ephemeris, attitude, sun vector, magnetic field vector, and other environmental data to simulators that are part of NOS³. 42 is open source C code. For NOS³ it has been packaged as a zip file which is installed on the virtual machine in the directory `/opt/42`. The STF-1 specific configuration files can be found in the directory `nos3/sims/cfg/InOut`. The main configuration files are the following:

1. `Inp_Sim.txt` – The main configuration file which defines items such as the environment (epoch, gravity models, celestial bodies, etc.), spacecraft reference orbits and configuration files, spacecraft and configuration files, and ground station locations.
2. `Orb_LEO.txt` – Spacecraft reference orbit file referred to by `Inp_Sim.txt`. This file specifies the orbit center (Earth) and refers to the two line element set file which defines the spacecraft orbit.
3. `SC_NOS3.txt` – Spacecraft definition file referred to by `Inp_Sim.txt`. This file defines labels, orbit parameters, initial attitude, body parameters, and other parameters specific to the spacecraft.
4. `Inp_IPC.txt` – File defining the TCP/IP or file parameters for communicating input and output to and from 42.



5. Inp_Graphics.txt – File defining the GUI configuration for 42, including what windows to display, parameters for the point of view, various display elements such as grids and labels, and other graphic elements properties.
 6. There are several other input files which are not used much for NOS³, including Inp_Cmd.txt (defining a command script for 42), Inp_FOV.txt (defining fields of view), Inp_Region.txt (defining regions for 42), and Inp_TDRS.txt (defining TDRS satellites for 42).
- c. As running AIT has previously been discussed, and is configured differently through yaml made telemetry pages, this section will focus on the use of COSMOS. COSMOS (GUI windows for Legal Agreement, COSMOS Command and Telemetry Server , Command Sender, Launcher)

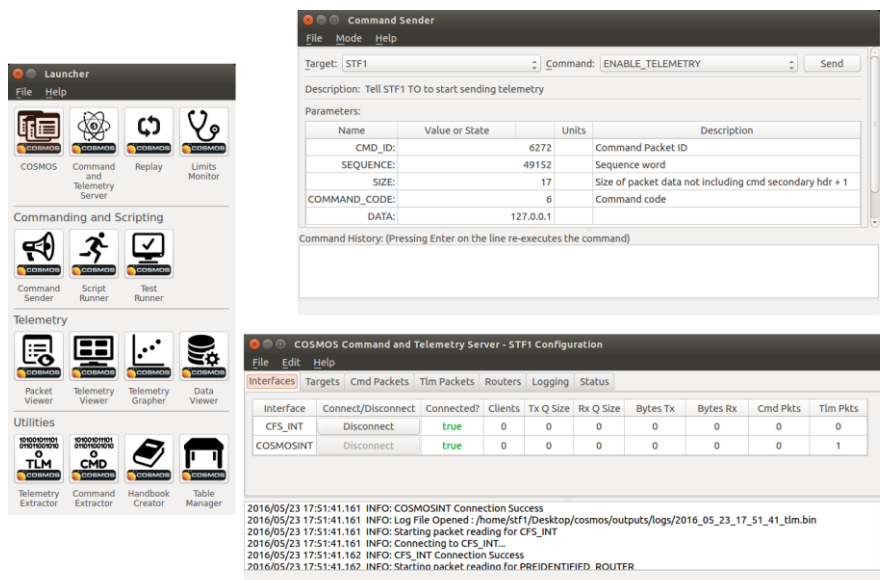


Figure 1723 - COSMOS

COSMOS is stated to be “The User Interface for Command and Control of Embedded Systems”. It is used by NOS³ as the ground station command and control system to send commands to and receive telemetry from the NOS³ flight software. COSMOS is installed as a Ruby Gem. The configuration for NOS³ has been created by configuration files which define the NOS³ “system” located *nos3/gsw/cosmos/config/system*

- d. Simulators (1 terminal window with a tab for each simulator, including the NOS Time Driver and the Simulator Terminal)



```

file edit View Search terminal tabs help
Magnetometer Simulator * GPS Simulator * EPS Simulator * Antenna Simulator * Cadet Simulator * Simulator Terminal * NOS Time Driver
2016-05-23 18:12:35.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1243, absolute time 52111643.000000
2016-05-23 18:12:36.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1244, absolute time 52111644.000000
2016-05-23 18:12:37.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1245, absolute time 52111645.000000
2016-05-23 18:12:38.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1246, absolute time 52111646.000000
2016-05-23 18:12:39.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1247, absolute time 52111647.000000
2016-05-23 18:12:40.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1248, absolute time 52111648.000000
2016-05-23 18:12:41.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1249, absolute time 52111649.000000
2016-05-23 18:12:42.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1250, absolute time 52111650.000000
2016-05-23 18:12:43.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1251, absolute time 52111651.000000
2016-05-23 18:12:44.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1252, absolute time 52111652.000000
2016-05-23 18:12:45.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1253, absolute time 52111653.000000
2016-05-23 18:12:46.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1254, absolute time 52111654.000000
2016-05-23 18:12:47.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1255, absolute time 52111655.000000
2016-05-23 18:12:48.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1256, absolute time 52111656.000000
2016-05-23 18:12:49.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1
157, absolute time 52111657.000000
2016-05-23 18:12:50.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1
158, absolute time 52111658.000000
2016-05-23 18:12:51.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1
159, absolute time 52111659.000000
2016-05-23 18:12:52.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1
160, absolute time 52111660.000000
2016-05-23 18:12:53.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1
161, absolute time 52111661.000000
2016-05-23 18:12:54.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1262, absolute time 52111662.000000
2016-05-23 18:12:55.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1263, absolute time 52111663.000000
2016-05-23 18:12:56.NF [ INFO ] Timedrvr::send_tick_to_nos_engine: tick = 1264, absolute time 52111664.000000
    
```

Figure 1824 - Simulators

Currently, NOS³ starts 8 simulators:

1. GPS Simulator – Simulates a hardware GPS using position and velocity data from 42.
2. EPS Simulator – Simulates a hardware electrical power system.
3. Battery Simulator – Simulates a cubesat battery.
4. CAM Simulator – Simulates a camera.
5. RW Simulator – Simulates a cubesat reaction wheel using momentum and torque data from 42.
6. Sample Simulator – Provides a basic example of a NOS³ simulator. The sample simulator assumes a UART based device. It streams telemetry at a fixed rate and recognizes a single configuration command to allow modification of the streaming rate.
7. Simulator Terminal – Provides a terminal to the nos3 user. This terminal can be used to send commands to other simulators on a special NOS Engine command bus and can be used to report data sent by the simulators on that special bus.
8. NOS Time Driver – This is the simulator component that provides the time source for NOS Engine. NOS Engine then distributes time to all clients that need it, including flight software and any simulator that needs to be aware of the passage of time in the simulated real world.

The simulators are all built and installed from source code as described in the previous section. The installation location is *nos3/sims/build/bin*. Various data and configuration files for the simulators can also be found in that location. Two of the main configuration files are as follows. The *sim_log_config.xml* file specifies the level and location of logging for the simulators. The *nos3-simulator.xml* file specifies the configuration for the simulators including common time, logging, and configuration information and information specific to each simulator. The specific information defines like the name of the simulator and if it is active, the hardware model (used to find the code plugin) for the simulator, the connection information (bus and name or address) for the simulator, and any environmental data provider information. The exact information for each simulator depends on the simulator, the hardware model, and potentially the data provider.



e. NOS³ Flight Software (1 terminal window)

```

NOS3 Flight Software
File Edit View Search Terminal Help
2015-012-14:03:20.27933 ES Startup: Loading file: /cf/sen.so, APP: SEN
2015-012-14:03:20.27980 ES Startup: SEN loaded and created
EVS Port1 55/1/FM 104: Free Space Table verify results: good entries = 3, bad =
0, unused = 5
EVS Port1 55/1/FM 1: Initialization complete: version 2.4.2.0
EVS Port1 55/1/FM 100: Child Task initialization complete
EVS Port1 55/1/CI_LAB_APP 6: CI: RESET command
EVS Port1 55/1/SCH 1: SCH Initialized. Version 2.2.0.0
EVS Port1 55/1/NAV 1: NAV Initialized. Version 1.0.0.0
EVS Port1 55/1/SC 21: RTS table file load count = 0
EVS Port1 55/1/CAM 4: CAM App: RESET Counters Command
EVS Port1 55/1/CAM 1: CAM App Initialized. Version 1.0.0.1
EVS Port1 55/1/SEN 4: SEN: RESET Counters Command
EVS Port1 55/1/SEN 1: SEN App Initialized. Version 1.0.0.0
EVS Port1 55/1/SC 9: SC Initialized. Version 2.4.0.0
EVS Port1 55/1/HK 1: HK Initialized. Version 2.4.0.0
EVS Port1 55/1/TO_LAB_APP 1: TO Lab Initialized. Version 2.1.0.0 Awaiting enable
command.
EVS Port1 55/1/CI_LAB_APP 3: CI Lab Initialized. Version 2.1.0.0
EVS Port1 55/1/EPS 1: EPS Initialized. Version 1.0.0.0
2015-012-14:03:20.38021 ES Startup: CFE_ES_Main entering OPERATIONAL state
EVS Port1 55/1/CFE_TIME 21: Stop FLYWHEEL
EVS Port1 55/1/SCH 18: Multiple slots processed: slot = 95, count = 5
    
```

Figure 1925 - NOS3 Flight Software

Last, but certainly not least is the NOS³ flight software. This is the flight software that will execute on the single board computer, but cross compiled to run on Linux and to use a hardware library that connects the flight software to the software only NOS Engine busses with their simulated hardware components instead of the actual flight hardware sensors and actuators.



9 NOS³ Workflows

Two workflows are common to utilize NOS³ as a user / developer:

1. Solely in the VM
2. Develop on host machine, test in VM

Both options make use of the vagrant virtual machine to provide a stable environment for testing. Out of the box it is assumed that option 1 is to be used. In order to switch to option 2, the following directions must be followed to properly configure the environment for use with the current scripts:

1. In the VM, go to Devices > Share Folders > Shared Folders Settings...

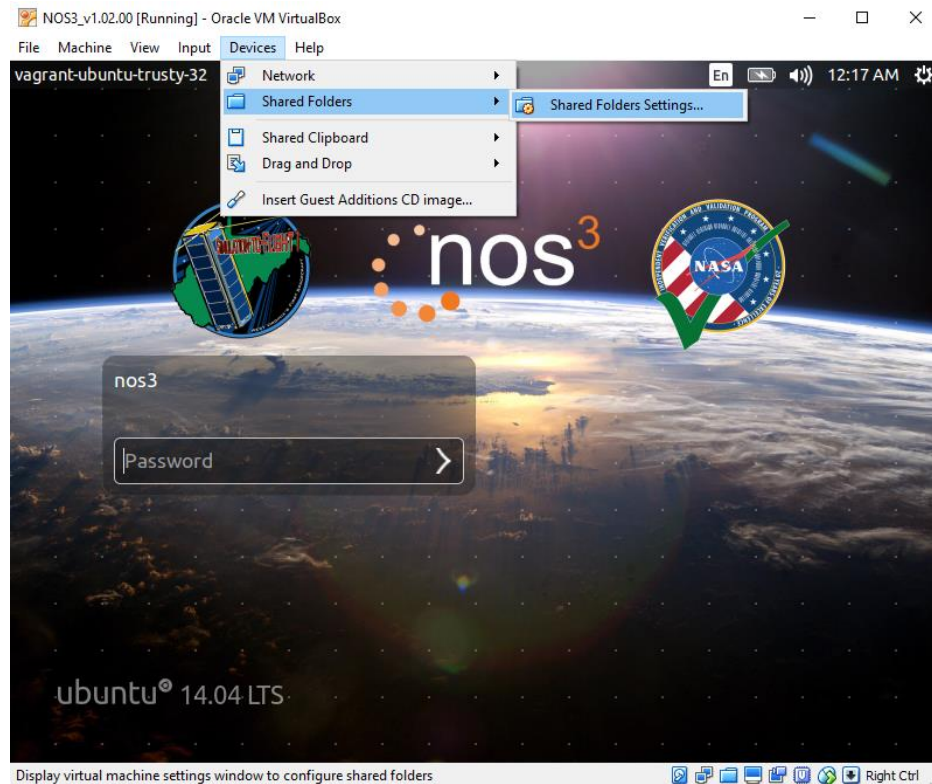


Figure 2026 - Shared Folder Settings

2. Add the 'nos3' folder cloned from git to the list of shared folders

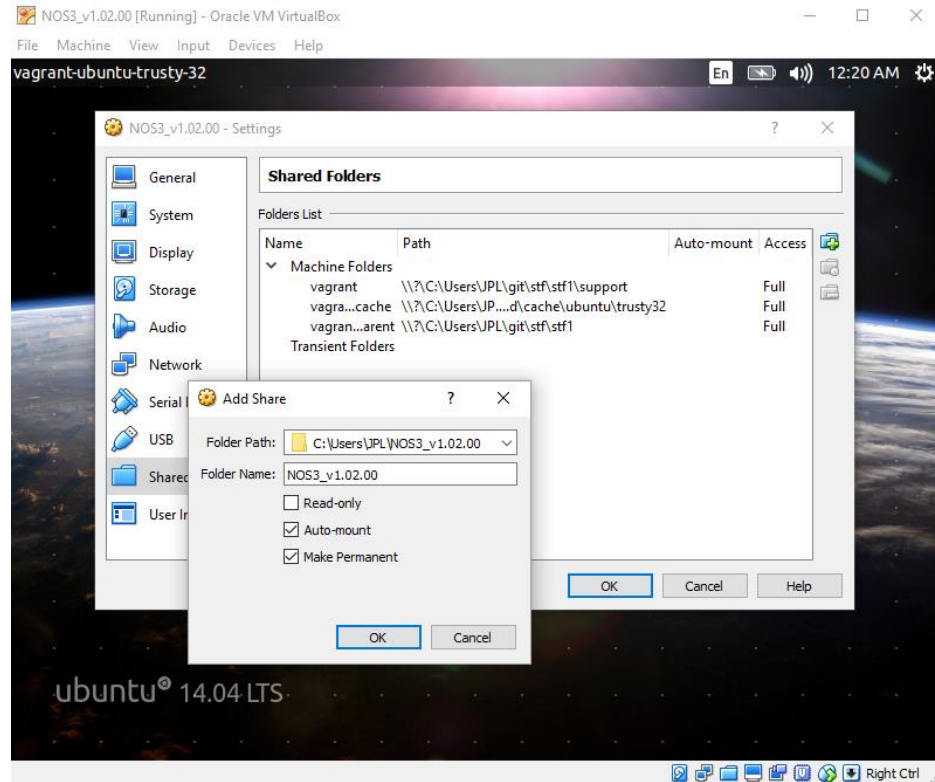


Figure 2127 - Share in NOS3 Folder

Once these steps are complete, all changes inside will be reflected outside and vice versa. If the VM is restarted, mounting the shared folder will need to be repeated.



10 Hardware Simulator Framework / Example Simulator

NOS³ simulator code has been developed in C++ with Boost and relies on the NASA Operational Simulator (NOS) Engine for providing the software busses, nodes, and other connections that simulate the hardware busses such as UART (universal asynchronous receiver/transmitter), I2C (Inter-Integrated Circuit), SPI (Serial Peripheral Interface), CAN (Controller Area Network), and discrete I/O (input/output) signals/connections/busses. NOS Engine also provides the mechanism to distribute time to all the simulators (and to the flight software).

10.1 Background and Supporting Concepts

10.1.1 Abstract Factory Design Pattern

C++ is a programming language that supports the Object Oriented programming paradigm, and within that paradigm, one of the most powerful design abstractions built on top of that paradigm are design patterns. The specific design pattern which has been heavily used within the NOS³ simulators to make them flexible and extensible is the Abstract Factory design pattern. This design pattern is described in many places, but one fairly easy to understand description is in the article “Abstract Factory Step-by-Step Implementation in C++” at <http://www.codeproject.com/Articles/751869/Abstract-Factory-Step-by-Step-Implementation-in-Cp>.

It is this factory design pattern that allows additional simulators to be easily constructed and built as plug-in libraries, even after the development of the initial NOS³ simulator code base. Instead of the shapes and shape factory in the article, the components in NOS³ simulators which are constructed via factories are hardware models and data providers.

10.1.2 XML Configuration

In addition to using the factory design pattern, each particular simulator must be configured to specify the hardware model to create. In addition, the hardware model may need parameters for configuring how the hardware acts. Also, hardware has connections for communication such as discrete I/O, I2C, or UART, and so in the simulation the hardware model will need to create software versions of these connections and these connections may also need configuration data such as bus type, bus name, and bus address. In addition, some hardware models (such as a GPS or magnetometer simulator) may need environmental data, and so the hardware model will need to create a data provider which will provide environmental data. The data provider may need configuration data such as the type of data provider and a filename or host and port.

The configuration for a specific simulation executable will be specified in a file via XML (eXtensible Markup Language), which will provide a list of simulators that are to be instantiated within that executable. Each simulator will specify a hardware model, which might have additional configuration parameters. The hardware model might specify reliance on an optional data provider with data provider configuration parameters. The hardware model might also specify one or more software communication connections with connection configuration parameters.



10.2 Implementing Your Own Hardware Model (and Data Provider, and Connections)

The following sections describe how to implement your own hardware model.

10.2.1 Configuration Data Property Tree

If configuration data from the XML file, which is represented as a configuration data property tree, is needed, it is retrieved using code like the following:

```
std::string param = config.get("simulator.<subname>.<subsubname>", "LITERAL");
```

The following are a few notes regarding this code. First, `config` is a variable of type `const boost::property_tree::ptree&`. Each hardware model and data provider must provide a constructor that takes a single parameter of this type (see below), and thus this parameter will be available to constructor code to perform any necessary configuration and initialization.

Second, when the code above is executed, the data type of the literal "LITERAL" determines the data type that the `ptree` tries to return your parameters as (here it is a literal string, and the variable the value is assigned to is declared accordingly as a `std::string`). Also note that you separate the XML tag names with periods in the key name to retrieve to indicate nested XML tag levels. Note also that you do not include the "nos3-configuration" or "simulators" prefixes in the key name (these appear in the default configuration file); they are stripped off by the `SimConfig` object which is used to read and parse the configuration data in the main program. Thus key names should either begin "common." or "simulator." If the key cannot be found in the property tree (which represents the XML), the value "LITERAL" is used as the default value.

The following is a list of common keys:

1. `common.log-config-file` – The name of the configuration file for logging using the ITC Logger class; you should not normally need to do anything with this.
2. `common.nos-connection-string` – Connect to nos time node ("`<commonkey>`", "`tcp://ip:port`")
3. `common.absolute-start-time` – The absolute start time of the simulation in decimal seconds from the J2000 epoch.
4. `common.sim-microseconds-per-tick` – The integer number of microseconds the simulation should advance for every time tick. Note that NOS Engine distributes time on its busses as a count of ticks. So if your hardware model or data provider receive the number of ticks that represents the simulation time, it can convert this to real world simulation time using:

```
double abs_time = _absolute_start_time + (double(ticks * _sim_microseconds_per_tick)) / 1000000.0;
```

5. `simulator.name` – The name you gave your simulator; it should agree with the string you put in the main function (see below).
6. `simulator.active` – Normally true; if false, then your simulator will not be run when the `SimConfig::run_simulator` method is called in the main function (see below).
7. `simulator.hardware-model.type` – The name string for your hardware model.
8. `simulator.hardware-model.connections` – A list of `<connection></connection>` tags which describes the connections that the hardware model has.



9. `simulator.hardware-model.data-provider` – Information on the data provider (if one is used and created using the data provider factory).
10. `simulator.hardware-model.data-provider.type` – The name string for your data provider (if one is used).

10.2.2 Hardware Model

The formula for creating a new hardware model is the following:

1. In namespace `Nos3`, create a class (e.g. `FooHardwareModel`) that inherits publicly from `SimIHardwareModel`.
2. Create a constructor that takes a `const boost::property_tree::ptree&` parameter which contains configuration data. Have the constructor retrieve configuration data and save any parameters and create any connections, data providers, or perform any other initialization that needs done for the hardware model.
3. Create a `void run(void)` method. This method should perform whatever tasks are supposed to be done when the hardware model is running.
4. Create a name string for your hardware model (e.g. `FOOHARDWARE`) and add a line like the following to your source file:

```
REGISTER_HARDWARE_MODEL(FooHardwareModel, "FOOHARDWARE");
```

5. If the hardware model uses a data provider, the hardware model could have a member variable of type `SimIDataProvider *`, which can be set in the hardware model constructor based on configuration data by lines like (assuming the member variable name is `_sim_data_provider`):

```
std::string dp_name = config.get("simulator.hardware-model.data-provider.type",
"BARPROVIDER");

_sim_data_provider = SimDataProviderFactory::Instance().Create(dp_name, config);
```

10.2.3 Data Provider

The formula for creating a new data provider is the following:

1. In namespace `Nos3`, create a class (e.g. `BarDataProvider`) that inherits publicly from `SimIDataProvider`.
2. Create a constructor that takes a `const boost::property_tree::ptree&` parameter which contains configuration data. Have the constructor retrieve configuration data and save any parameters or do any initialization that needs done for the data provider.
3. Create a `virtual boost::shared_ptr<SimIDataPoint> get_data_point(void) const;` method... that does whatever is supposed to be done to retrieve (or compute or whatever) a data point when your data provider is asked for a data point and which returns a pointer to the retrieved data point. You should also create a class that inherits publicly from `SimIDataPoint` to hold the data that you return from the data provider.
4. Create a name string for your data provider (e.g. `BARPROVIDER`) and add a line like the following to your source file:

```
REGISTER_DATA_PROVIDER(BarDataProvider, "BARPROVIDER");
```



10.2.4 Connections

The general procedure for creating a connection is to create an object that is called a hub (a default constructed object can be used), then create bus and node objects or a connection object (depending on the connection type). With the node or connection object, various things can be done to handle the connection such as registering a callback so that when a message is received on the connection, the hardware model can respond to it and send a response. The basics for using a few of the connection types are described below, but for examples, please consult the example code and existing simulators.

10.2.4.1 Command Connection

The command connection of a simulation hardware model is not a normal connection in the sense of a connection that the hardware would have to a hardware bus. It is used just to perform out of band commanding of the simulation itself. One way to perform this commanding is to use the SimTerminal executable that is part of NOS³. This terminal starts up and registers as a node on the command bus. It can then be used to send messages to any other node on the command bus. These messages can be ASCII or hexadecimal bytes.

The base `SimIHardwareModel` creates a node on a command bus so that any hardware model simulation can be commanded. In order for a simulation to perform actions based on commands received on the command bus, the only thing that needs done in the hardware model is the following:

1. In the hardware model class, override the `SimIHardwareModel` method:

```
void command_callback(NosEngine::Common::Message msg)
```

For an example of how data is received by and returned from the hardware model in response to a command, refer to the `command_callback` method in the base `SimIHardwareModel` class.

10.2.4.2 Time Connection

For the hardware simulator to have a notion of time in the real world, it registers a node with NOS Engine as a time client node. The formula for creating and using a time client node is:

1. In the hardware model class, add member variables for the bus and time node, e.g.:

```
std::unique_ptr<NosEngine::Client::Bus> _time_bus;
NosEngine::Client::TimeClient*      _time_node;
```

2. In the hardware model constructor:

- a. The base `SimIHardwareModel` class has an existing hub, member variable `_hub` for the bus to connect to. The connection string for NOS Engine can be retrieved from the XML configuration data by a call like:

```
std::string connection_string = config.get("common.nos-connection-string", "tcp://127.0.0.1:12001");
```

- b. Add a "time" type connection to the XML configuration file something like:

```
<connection><type>time</type><bus-name>command</bus-name><node-name>my-time-node</node-name></connection>
```

- c. Retrieve the bus name and node name into `std::string` variables like `time_bus_name` and `time_node_name`. For an example of how to do so, please see the example simulator.



- d. Create a bus object:


```
_time_bus.reset(new NosEngine::Client::Bus(_hub,
connection_string, time_bus_name));
```
- e. Create a time client node on the bus:


```
_time_node = _time_bus->get_or_create_time_client(time_node_name);
```
3. In hardware model methods that need time:
 - a. To get the number of “ticks” that have elapsed, call:


```
_time_node->get_last_time()
```
 - b. To convert this to real world time, the `SimIHardwareModel` has member variables `_absolute_start_time` and `_sim_microseconds_per_tick` (set from data in the common section of the XML configuration file), and they can be used to compute real world time by:


```
_absolute_start_time + (double(_time_node->get_last_time() *
_sim_microseconds_per_tick)) / 1000000.0);
```
4. To clean up, in the hardware model destructor, call:


```
_time_bus.reset();
```

10.2.4.3 UART Connection

For hardware that is connected via UART, the formula for the hardware to creating and using a node on the UART bus is the following:

1. In the hardware model class, add a member variable for the UART connection like the following:


```
std::unique_ptr<NosEngine::Uart::Uart> _uart_connection;
```
2. In the hardware model constructor:
 - a. The base `SimIHardwareModel` class has an existing hub, member variable `_hub` for the bus to connect to. The connection string for NOS Engine can be retrieved from the XML configuration data by a call like:


```
std::string connection_string = config.get("common.nos-connection-string", "tcp://127.0.0.1:12001");
```
 - b. Add a “usart” type connection to the XML configuration file something like:


```
<connection><type>usart</type><bus-name>usart_0</bus-name><node-port>99999</node-port></connection>
```
 - c. Retrieve the bus name and node port into `std::string` variables like `bus_name` and `node_port`. For an example of how to do so, please see the example simulator.
 - d. Create a UART connection object:


```
_uart_connection.reset(new NosEngine::Uart::Uart(_hub,
config.get("simulator.name", "foosim"), connection_string,
bus_name));
```
 - e. Open the connection and set a callback for when the hardware UART is read:


```
_uart_connection->open(node_port);
_uart_connection->set_read_callback(
std::bind(&FooHardwareModel::uart_read_callback,
this, std::placeholders::_1, std::placeholders::_2));
```



3. Create a hardware model method for the callback (here is where most of the custom work for a specific hardware model would be done):
 - a. The signature should be like:


```
void FooHardwareModel::uart_read_callback(const uint8_t *buf, size_t len);
```
 - b. To return data, use the UART method:


```
size_t UART::write(const uint8_t *const buf, size_t len);
```
 - c. For an example, consult the example sim code.
4. In the hardware model destructor, make the call:


```
_uart_connection->close();
```

10.3 Writing Your Own Simulator

The following formula describes how to create a simulator using a hardware model (and optionally a data provider) created using the formulas above:

1. Create a main source file with the following contents:

```
#include <ItcLogger/Logger.hpp>
#include <sim_config.hpp>

namespace Nos3
{
    ItcLogger::Logger *sim_logger;
}

int
main(int argc, char *argv[])
{
    std::string simulator_name = "foosim"; // this is the ONLY simulator specific line!

    // Determine the configuration and run the simulator
    Nos3::SimConfig sc(argc, argv);
    Nos3::sim_logger->info("main:  %s simulator starting",
        simulator_name.c_str());
    sc.run_simulator(simulator_name);
    Nos3::sim_logger->info("main:  %s simulator terminating",
        simulator_name.c_str());
}
```

2. Change "foosim" to whatever you would like the name of your simulator to be
3. Add XML like the following inside the <simulators></simulators> tags in the standard configuration file (the standard configuration file name is nos3-simulator.xml)

```
<simulator>
  <name>foosim</name>
  <active>true</active>
  <library>libexample_sim.so</library>
  <hardware-model>
    <type>FOOHARDWARE</type>
    <connections>
      <connection>
        <connection-param1>cp1</connection-param1>
        <!-- ... -->
        <connection-paramN>cpN</connection-paramN>
      </connection>
    </connections>
  <data-provider>
    <type>FOOPROVIDER</type>
    <provider-param1>fppl</provider-param1>
```



```

        <!-- ... -->
        <provider-paramN>fppN</provider-paramN>
    </data-provider>
    <other-hardware-parameter1>OTHER-FOO</other-hardware-parameter1>
    <!-- ... -->
    <other-hardware-parameterN>OTHER-FOO</other-hardware-parameterN>
</hardware-model>
</simulator>

```

4. Customizing the XML:

- a. The `simulator.name` should be the same as in your main function in #1.
- b. The `simulator.active` tag should be true unless you do not want your simulator to run in which case it should be false.
- c. The `simulator.library` tag should contain the name of the example simulator shared object library file (normally `lib<project>.so` where `<project>` is the project name given the project in the `CMakeLists.txt` file; see below)
- d. The `simulator.hardware-model.type` should be the same as the string you used in the `REGISTER_HARDWARE_MODEL` line above.
- e. The simulator `hardware-model` `data-provider` type should be the same as the string you used in the `REGISTER_DATA_PROVIDER` line above.
- f. All other tags are up to you... create your own names and then use the information above for accessing the data. Note that there are examples in the source code for using several common connection types such as UART, I2C and the command connection (used to control the simulator with the simulator terminal). Also note that the command connection is automatically configured for you in the `SimIHardwareModel` base class. To have your simulator respond to commands to it on the command bus, all you need to do is override the `SimIHardwareModel::command_callback` method in your hardware model class (the default implementation does nothing).



10.4 Example Simulator

Hopefully this introduction is useful in describing the flexible, extensible framework employed in developing NOS³ simulators. This introduction has attempted to describe the design pattern used within NOS³ simulators and described how to add hardware models (and data providers and other supporting items), and put hardware models together into standalone simulators that can be part of the NOS³ simulation environment.

For a complete example, refer to the source code and CMakeLists.txt file in the nos3 git repository, subdirectory sims/sample_sim/ and refer to the configuration file in the nos3 git repository, file sims/cfg/nos3-simulator.xml (see the simulator section with name "sample_sim"). Note also that if a new simulator's CMakeLists.txt file for a simulator has a project name line like "project(sample_sim)" at the beginning, the line "add_subdirectory(sample_sim)" may be added under # NOS3 Sim Core in the sims/CMakeLists.txt file in the nos3 git repository so that the new simulator will be built, but the sims/CMakeLists.txt file is written to find all properly structured and properly named directories following the form of the parent folder in nos3/sims/ being: "<name-of-your-sim>_sim".



11 42, A Visualization and Simulation Tool for Spacecraft Orbit and Attitude Dynamics

11.1 42 Overview

Some of the simulated hardware components require dynamic environmental data. 42 is an open source visualization and simulation tool for spacecraft attitude and orbital dynamics and environmental data developed by NASA's Goddard Space Flight Center (GSFC). The role of 42 within NOS³ is to provide dynamic environmental data required by the simulated hardware components.

The presentation material on 42 describes it as a general-purpose, multi-body, multi-spacecraft simulation. The presentation materials describe the following features of 42 which are of interest to NOS³ (other features are described as well):

1. Multiple spacecraft, anywhere in the solar system
 - a. Two-body, three-body orbit dynamics (with seamless transition between)
 - b. One sun, nine planets, 45 major moons

The presentation materials also list the following environmental models which are of interest to NOS³ (other models are described as well):

1. Planetary Ephemerides
 - a. From Meeus, "Astronomical Algorithms"
 - b. Good enough for GNC validation, not intended for mission planning
2. Gravity Models have coefficients up to 18th order and degree
 - a. Earth: EGM96
3. Planetary Magnetic Field Models
 - a. IGRF up to 10th order (Earth only)
4. Earth Atmospheric Density Models
 - a. MSIS-86 (thanks to John Downing)
 - b. Jacchia-Roberts Atmospheric Density Model (NASA SP-8021)

42 uses a collection of input files to control its execution. For NOS³, the main configuration files of interest are the following:

7. Inp_Sim.txt – The main configuration file which defines items such as the environment (epoch, gravity models, celestial bodies, etc.), spacecraft reference orbits and configuration files, spacecraft and configuration files, and ground station locations.
8. Orb_LEO.txt – Spacecraft reference orbit file referred to by Inp_Sim.txt. This file specifies the orbit center (Earth) and refers to the two line element set file which defines the spacecraft orbit.
9. SC_NOS3.txt – Spacecraft definition file referred to by Inp_Sim.txt. This file defines labels, orbit parameters, initial attitude, body parameters, and other parameters specific to the spacecraft.
10. Inp_IPC.txt – File defining the TCP/IP or file parameters for communicating input and output to and from 42.
11. Inp_Graphics.txt – File defining the GUI configuration for 42, including what windows to display, parameters for the point of view, various display elements such as grids and labels, and other graphic elements properties.



12. There are several other input files which are not used much for NOS³, including `Inp_Cmd.txt` (defining a command script for 42), `Inp_FOV.txt` (defining fields of view), `Inp_Region.txt` (defining regions for 42), and `Inp_TDRS.txt` (defining TDRS satellites for 42).

11.2 Providing Data to a Simulator from 42

When 42 is run, it writes environmental data to a set of files that have the extension “.42”. The data written in the “MAG.42” file can be used by the `MagnetometerSimDataFileProvider` data provider while the data written in the “FOTON.42” file can be used by the `GPSSimDataFileProvider`.

In addition to using files of 42 data, 42 can output data to a TCP/IP socket. This output is controlled by the input file “`Inp_IPC.txt`”. To output data to a TCP/IP socket and act as a server (the mode used by NOS³ hardware simulator data providers such as `MagnetometerSimData42SocketProvider` and `GPSSimData42SocketProvider`), the “IPC Mode” should be set to “TX”, the “Socket Role” should be set to “SERVER” and the “Server Host Name, Port” should be set to the host name or IP address to use and the TCP socket port number to use.

11.3 Coordinating 42 Time

When data is output to a TCP/IP socket and in order to maintain a consistent real time reference within the system, 42 has been modified so that it can have its time driven by NOS Engine (which also drives hardware simulator and flight software time as well). To configure 42 to use NOS Engine driven time, the “`Inp_Sim.txt`” and “`Inp_NOS3.txt`” files are modified as follows. In “`Inp_Sim.txt`” change the “Time Mode” line to have the value “NOS3”. In “`Inp_NOS3.txt`” set the “NOS3 Time Connection String” line to have the connection string for contacting the NOS Engine Standalone Server and set the “Sim Time Bus” line to the NOS Engine bus name to use to retrieve time from.



11.4 Data Available from 42

The following data is currently written by 42 to the TCP/IP socket and can be used as environmental data for data providers: date/time, spacecraft in eclipse/sunlight, spacecraft position in the inertial world frame, direction cosine matrix for conversion from inertial world frame to rotating world frame, spacecraft position in the rotating world frame, spacecraft velocity in the inertial world frame, direction cosine matrix for conversion from spacecraft inertial frame to spacecraft body frame, spacecraft angular velocity, quaternion for conversion from spacecraft inertial frame to spacecraft body frame, vector from spacecraft to sun in the inertial world frame, magnetic field vector at the spacecraft in the inertial world frame, and spacecraft angular momentum.



12 Flight Software Development, Especially Using cFS

The preferred operating system for use with NOS³ is the open-source Core Flight System (cFS) originally developed by NASA GSFC. This section will describe the method utilized to interface NOS³ with cFS, as well as a generic method to interface with any flight software that can compile for Linux.

12.1 cFS and NOS³

12.1.1 Operating System Abstraction Layer

Core Flight System is the FSW selected for the STF-1 mission partially due to the implementation of the Operating System Abstraction Layer (OSAL). The OSAL provides an API that allows flight software applications to be written without operating system (OS) specific calls. When cFS is compiled, the target OS is specified and the build system includes the proper libraries. This allows the FSW written for the FreeRTOS target to be built to execute on Linux and the opposite remains true. This makes NOS³ an ideal development environment when using the OSAL Linux target.

12.1.2 Platform Support Package

In addition to the OSAL, cFS includes a Platform Support Package (PSP) that includes libraries that are not OS specific, but can be reused for a specific flight board, such as memory, clocks, timers, etc. The PSP used for NOS³ is a modified version of the Linux PSP release. In order to control timing in flight software, cFS uses multiple timers, the main being a 1 Hz timer tick. By replacing the 1 Hz timer provided by Linux with the NOS Engine time ticker, we can sync the time from the PSP, with the time that other NOS³ components are running.

12.1.3 Hardware Library

The third component of flight software implemented for hardware abstraction is a hardware library (HWLIB). The HWLIB is used for component specific I/O calls, such as I2C, UART, etc. The hardware library includes a single header file, typically provided as drivers from the on-board computer (OBC) manufacturer, that define the I/O function calls. When building cFS, the CMAKE build system then selects the driver source corresponding to the target being built.

As an example, the Clyde Space EPS I/O functionality is well defined in the user's manual, and communications are performed over I2C. Using the NanoMind (STF-1 OBC) I2C drivers, a library called epslib.c is written to communicate over I2C and exercise all of the EPS functionality as described in its documentation. When compiling for the flight target, the NanoMind driver source code is selected by CMAKE and the executable can be run on the OBC. When compiling for Linux, the CMAKE build will select the NOS³ driver source code and the executable can be run in the NOS³ environment. With either path, the HWLIB and all code using the HWLIB will remain unchanged, and only the low level drivers will be effected. The diagram below shows the two path example as it applies to STF-1, where LIBA3200 is the NanoMind source, and LIBA3200NOS is the NOS³ source.

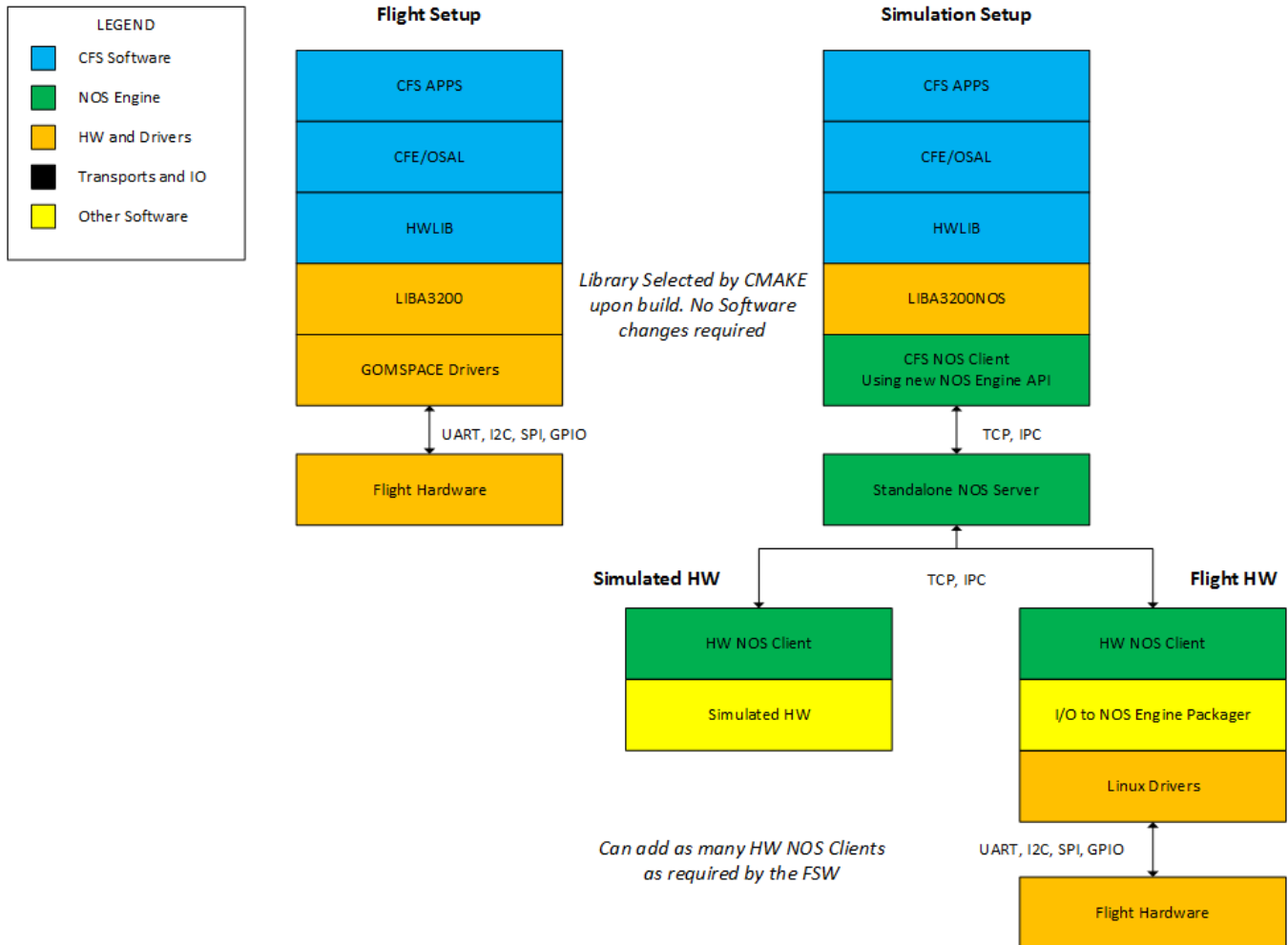


Figure 2228 - Flight and Simulation Targets

12.2 Connecting cFS to NOS³

In order to use NOS³ with cFS, modifications are required to the open-source release. The recommended method for using NOS³ is described in the NOS³ User's Guide, in which these modifications have already been made. If not using the cFS included with the NOS³ release, it is recommended to use the CMAKE build system, as the legacy build is not currently supported. The necessary changes are described below, where "proj" is the cFS directory being integrated.

1. Edit the *targets.cmake* file in the *nos3/fsw/nos3_defs* folder to include the list of applications to be built. Set the target name and system as shown below.

```
SET(TGT1_NAME linux)
SET(TGT1_SYSTEM linux)
```

2. Edit the *toolchain-linux.cmake* from the *nos3/fsw/nos3_defs* directory .
3. Edit the *nos-linux* PSP from the *nos3/fsw/psp/fsw* directory.
4. Add to the *apps* directory in *nos3/fsw/apps* as needed.



5. Create a `nos3/fsw/components/hwlib` or edit the `nos3/fsw/components/hwlib` directory.
 - a. The `CMakeLists.txt` file in `nos3/fsw/components/hwlib` will provide a good example of how to include driver source code as described in section 6.1.3.
 - b. Add a `sim` folder to this directory to store the NOS³ drivers for I/O. (See section 6.3 for an example driver)
6. Edit or Add the needed simulation components in the `nos3/sims` directory.

12.3 NOS³ Drivers and Other FSW

It is possible to connect NOS³ to FSWs other than cFS, although this has not been extensively tested. The two main requirements are the availability of source code for the I/O drivers, and the ability to compile/run on Linux. If these two conditions are met, the drivers for the target hardware can be swapped for NOS³ drivers as described in previous sections.

12.3.1 Writing a NOS³ Driver

The NOS³ source is the best resource for examples to aid in writing a new NOS³ driver. The UART driver and STF-1 NAV (navigation) application will be used in the example described in this section. For this example, the NAV application is written for cFS and located in the `novatel_oem615` component folder, but this application could just as easily be any other FSW source file.

12.3.1.1 Application and Hardware Library

The application that is communicating with hardware will require the I/O calls to be implemented exactly as provided by the OBC manufacturer. The NAV application makes certain calls to a Novatel GPS over the UART from the OBC. Not all of the GPS functionality is necessary to be exercised by the NAV application, so the low level calls to the UART are wrapped in functions in the hardware library, and the NAV app includes this library. As an example, the NAV application will be commanded to get the current Position/Velocity/Time reading, and will make the call `NAV_ReadAvailableData` as seen in the following code excerpt. Notice the include statement for the hardware library.

```
#include "hwlib.h"

/* some code removed for readability see nos3/fsw/components/novatel_oem615/fsw/src/nav_app.c */

/* Request NAV data */
case NAV_REQ_DATA_CC:

CFE_EVS_SendEvent(NAV_CMD_REQ_DATA_EID, CFE_EVS_DEBUG, "Request NAV GPS Data");

/* todo - fix the 1024 hard coded number */
DataBuffer = (uint8_t *)malloc((1024) * sizeof(uint8_t));

/* Read the GPS data from the UART */
NAV_ReadAvailableData(DataBuffer, &DataLen);

GPSSerialiation GPSData = NAV_ParseOEM615Bestxyza(DataBuffer, DataLen);
```

The function `NAV_ReadAvailableData` is a wrapper for the low level UART calls to the OBC driver. The function can be seen in the following code excerpt. This code must include the hardware library, as seen in the first line of the excerpt, which includes the OBC drivers itself. The bold function calls are from the OBC driver.



```
#include "hwlib.h"

/* some code removed for readability see nos3/fsw/components/novatel_oem615/fsw/src/nav_app.c */

static void NAV_ReadAvailableData(uint8 DataBuffer[], int32 *DataLen)
{
    int32 i = 0;

    /* check how many bytes are waiting on the uart */
    *DataLen = uart_bytes_available(NAV_UART.handle);
    //OS_printf("NAV_ReadAvailableData(): gps messages waiting: %ld bytes\n", (long
int)*DataLen);

    /* declare an out buffer to hold that data */
    if (*DataLen > 0)
    {
        /* grab the bytes */
        uart_read_port(NAV_UART.handle, DataBuffer, *DataLen);
    }
    else
    {
        /* OS_printf("GPS_ReadAvailableData(): gps uart data len is 0\n"); */
    }
}
```

12.3.1.2 The NOS³ Driver

The example described above uses the hwlib.h header which provides the device drivers for the Novatel_OEM615. This header is included by any library making calls to the UART and can be stored at any location. In this case both the hardware library and device drivers are located in *nos3/fsw/components/hwlib/fsw/public_inc/*.

The functions used by the GPS app in this example are *uart_bytes_available* and *uart_read_port*, which are defined in *nos3/fsw/components/hwlib/sim/src/libuart.c*. The *uart_bytes_available* function will be examined in more detail for this example. The code excerpt below shows the function.

```
/* usart number bytes available */
int32 uart_bytes_available(int32 handle)
{
    int bytes = 0;
    NE_Uart *dev = nos_get_usart_device((int)handle);
    if(dev)
    {
        OS_MutSemTake(nos_usart_mutex);
        bytes = (int)NE_uart_available(dev);
        OS_MutSemGive(nos_usart_mutex);
    }
    return bytes;
}
```

The function above is used to return the number of bytes available from the USART buffer to the calling function. The *NE_uart_available* function used in this code is provided by NOS Engine (reference section 3). Details about the UART, I2C, and SPI NOS plugins can be found in the NOS Engine user's manual.

12.3.1.3 Build System

The build system must be able to properly select the correct driver source code based on the target being compiled. In this case, CMake is used by both cFS and NOS³ and can accomplish this swap easily. As



described in section 6.2 the *targets.cmake* file in *nos3/fsw/nos3_defs* will an example of how to include driver source code; an example Cmake build may be found in *nos3/fsw/components/novatel_oem615/CmakeList.txt*.



13 Hardware In The Loop

Hardware in the Loop functionality is being reevaluated as a feature to include in future releases.



14 Orbit, Inview, and Power Planning Tool

Several planning tools are envisioned to be created for STF-1 mission operations. The first is the Orbit, Inview, and Power Planning tool. The role of OIPP will be to execute daily and perform the following tasks:

1. Retrieve the most up to date two-line element set (TLE) data string for the STF-1 CubeSat,
2. Propagate this element set forward for a number of days in the future, compute in view periods with STF-1 ground antennas (nominally only NASA Wallops) for a number of days in the future, and determine sunlight and eclipse periods for STF-1 for a number of days.

It should be noted that the accuracy of all predictions deteriorates as the propagation is performed further into the future, thus the most accurate data will typically be for the first day in the future predictions and the least accurate data will typically be for the last day in the future predictions. Thus, the later future data is used for approximate planning, while the near future data is used for upcoming day(s) operations.

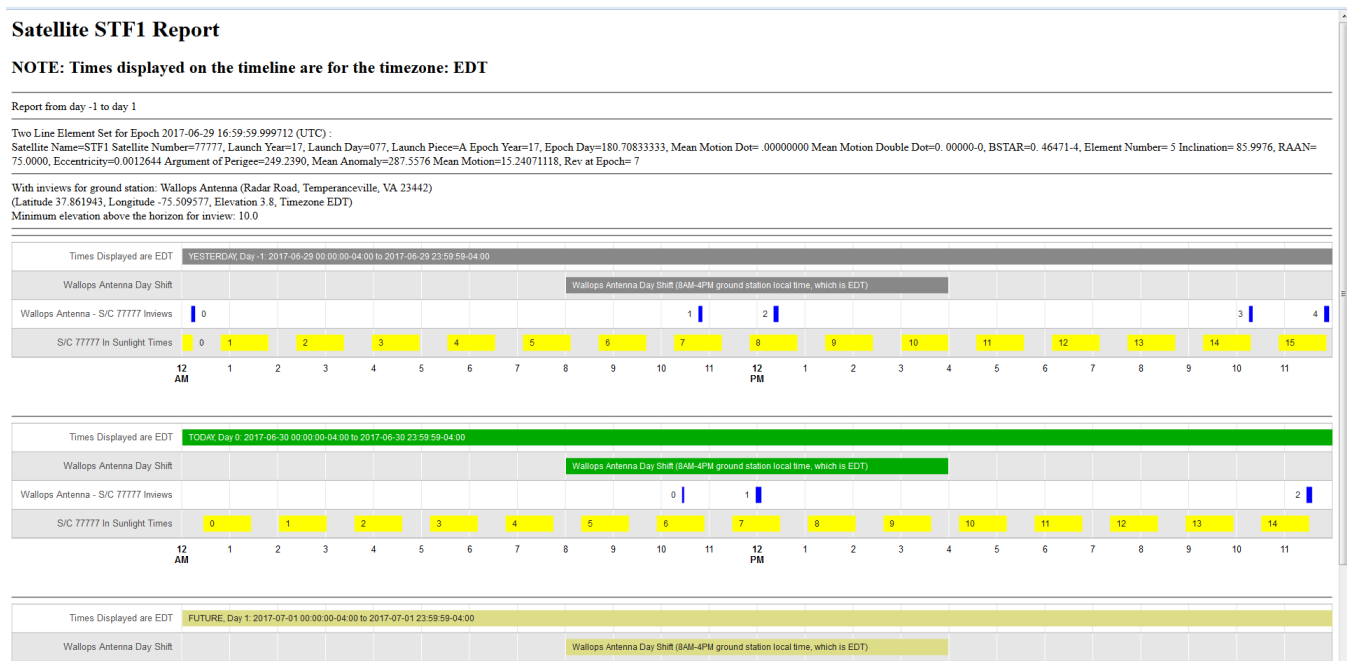


Figure 2329 - Example OIPP Report

A link to execute OIPP in the NOS3 VM can be found on the desktop at `"stf1-oipp-demo.sh"`. Double clicking the script will run for a while, generating the report `"stf1-oipp.html"` on the desktop, which will then be displayed similar to what is shown above in a web browser. The tool can be found in the directory `"/home/nos3/Desktop/planning/OrbitInviewPowerPrediction"`. For the demo version, the TLE that is used is the same one that is used by 42 and is symbolically linked in the directory `"/home/nos3/Desktop/planning"`.
 QUICK-START : `python ./generate_html_report.py`

```

Read the README.MD
sudo -H pip install pytz
sudo -H pip install pyorbital
sudo -H pip install geocoder
sudo -H pip install openpyxl
    
```



```
sudo -H pip install python-dateutil
sudo -H pip install matplotlib
```

15 FAQ

- How do I logon?
 - User: Vagrant Password: Vagrant
 - User: nos3 Password: nos3123!
- NOS Engine bus ports fail on launch
 - NOS Engine allows dynamic connections and disconnects and ensures ports are closed before connecting. Check to make sure, the ports may work again after an initial “Not connect.”
- When the cFS FSW starts it cannot find my application or startup script
 - Make sure that your application is built correctly and the .so is present. Likewise, ensure that the app name is correctly listed in *nos3/fsw/nos3_defs/cfe_es_startup.scr* and *nos3/fsw/nos3_defs/targets.cmake*.
 - For further information, please check the NASA/cFS Git Repository and documentation.
- How do I connect my own standalone FSW?
 - Be sure to have all port numbers consistent between all components in NOS³, including 42.
- Why does cFS constantly crash on start-up and/or force me to restart my PC to rerun?
 - NASA’s cFS is safety-critical flight software, make sure you are building your applications to specification and that you are properly using the PSP and OSAL calls from within your apps.
 - It is best to **not** run cFS as sudo. If you are doing this, make sure you have configured for your host or are providing appropriate run-time arguments with cFS.