NASA Operational

Simulator for Small Satellites

nos3

User Manual

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

Table of Figures

1 Introduction

This document, titled "NOS^3 User Manual", provides information for users and developers that intend to enhance and extend the NASA Operational Simulator for Small Satellites (NOS^3).

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 NOS^3.

NOS^3 is an open-source, software only test bed for small satellites with the intention of becoming NASA Open Source in the future. 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).

NOS^3 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
Oracle VirtualBox and Vagrant	Oracle VirtualBox is an open source solution for creating and running virtual machines. 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.
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	The third component is 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 used as the base system which STF-1 flight software is developed on top of.
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.
CFC	The COSMOS File Creator (CFC) allows for the generation of command and telemetry files from FSW, barring it contains the proper comments to be parsed.

1.2 Format

The format of this document is as follows. Section 2 describes the overall architecture of NOS^3, 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 is a quick start guide with a minimum set of procedures for creating the NOS^3 virtual machine, building the flight software and simulator components on the NOS^3 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 5 provides detailed instructions for creating the NOS^3 virtual machine and describes the steps that occur in the Vagrantfile to configure and provision the virtual machine.

Section 6 provides detailed instructions for building the flight software and simulations on the NOS^3 virtual machine.

Section 7 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 8 elaborates on the various types of NOS^3 workflows that exist. This includes setting NOS^3 for editing and using version control on your host or inside of the VM.

Section 9 describes the framework for developers to use to develop hardware simulators and provides information on example simulator code included with NOS^3.

Section 10 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 11 describes developing flight software using cFS and interfacing it with NOS engine and the simulators.

Section 12 describes the COSMOS ground system and how it 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 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 NOS^3 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 only 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 are installed including 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 or utilize the same provisioner script on an Ubuntu 16.04 virtual machine / host; however, to build and run the core flight software, simulators, and so on, the source code will still need to be present as describe below.

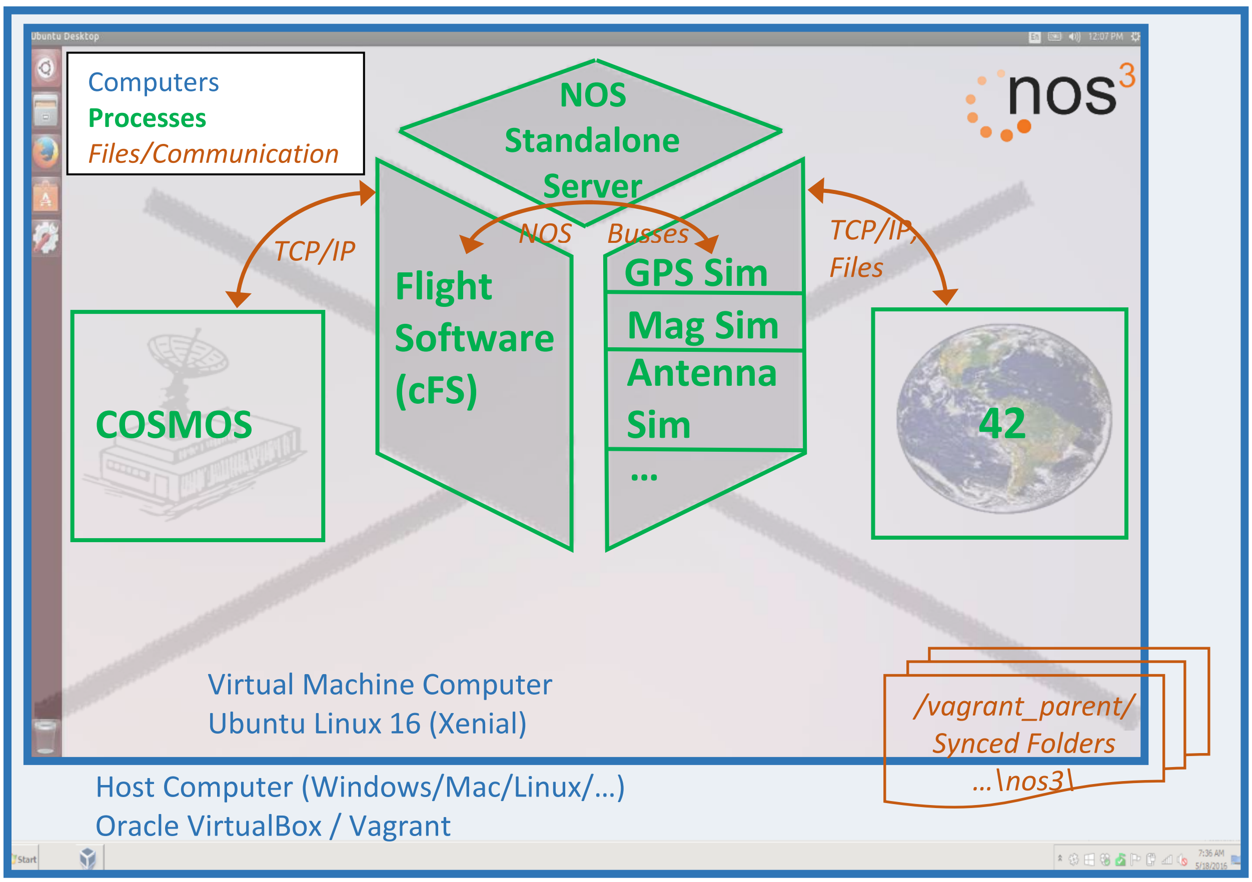


Figure 1 - NOS³ Architecture

Finally, source code for various simulators is present on the virtual machine through synced 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 these 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 which 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 a separate NOS engine command bus which all of the simulators can be nodes on.

In addition, the cFS source code is also present on the virtual machine through synced folders. Build tools can be used to build and install the generic flight software also. 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. 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, formally referred to as a bus. A bus can have an arbitrary number of nodes, and each node within the bus must have a name that is unique to all 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 NOS^3, NOS Engine is used to provide software simulations of the hardware buses. It provides the infrastructure for each hardware simulator to be a node on the proper bus and for the flight software to interact with the hardware simulator nodes on the appropriate bus. NOS Engine provides plug-ins for various protocols such as MIL-STD-1553, SpaceWire, I2C, SPI, and UART. These plug-ins allow each bus and the nodes on the bus to communicate using calls and concepts that are specific to that protocol.

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

4 Quick Start to Installing, Building, and Running NOS3

4.1 Installing NOS3

On the host computer:

1. Install Oracle VirtualBox (https://www.virtualbox.org/)
2. Install Vagrant (https://www.vagrantup.com/)
3. Acquire the *nos3* release
4. Unzip the package into your preferred working directory
5. Execute the installation script for your specific host environment
    a. Windows and Linux and currently supported
6. A configuration menu will appear:

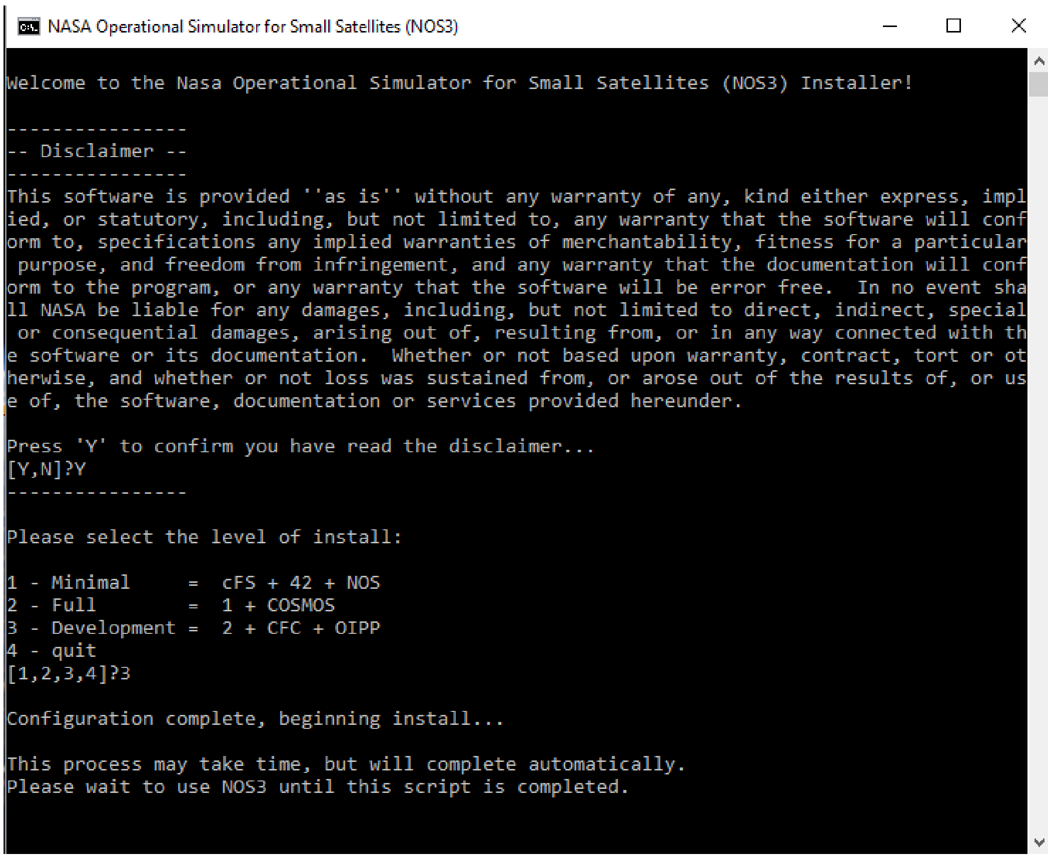


Figure 2 - NOS3 Installation Configuration Menu

7. Once completed, allow the installer to finish its work. This will take some time, anywhere from ½ hour to an hour or more for the first run. **IMPORTANT:** Internet access is **required** when running. Also, please do **NOT** use the virtual machine until the installation process is complete.

Figure 3 - NOS3 Virtual Machine Complete

4.2 Building NOS^3

Log in to the NOS3 VM: *nos3 / nos3123!*

1. Double click "***nos3-build.sh***"
2. A terminal window will come up and build the simulator and flight software (this will take quite a few minutes)

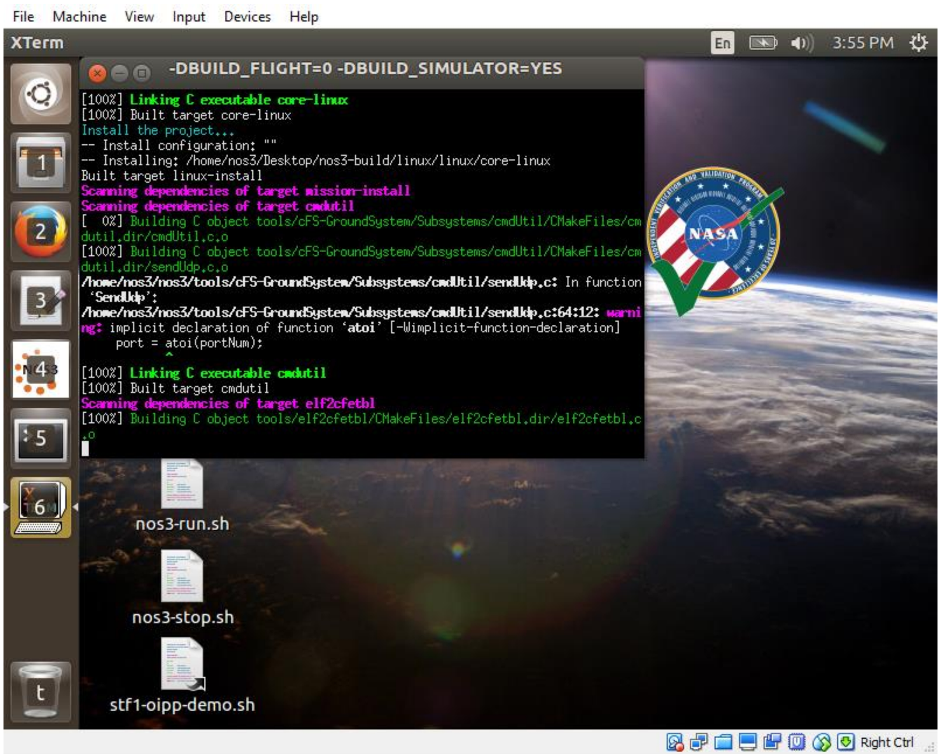


Figure 4 - Building Simulators and Flight Software

4.3 Running NOS³

Log in to the NOS3 VM: *nos3 / nos3123!*

1. Double click "**nos3-run.sh**". The following software will start up:
    a. NOS Engine Standalone Server (1 terminal window)
    b. 42 Dynamic Simulator (1 terminal window, 1 GUI window with CubeSat, 1 GUI window with map)
    c. COSMOS (GUI windows for Legal Agreement, COSMOS Command and Telemetry Server – STF1 Configuration, Command Sender, Launcher)
    d. Simulators (1 terminal window with a tab for each simulator, including the NOS Time Driver and the Simulator Terminal)
    e. STF1 Flight Software (1 terminal window)

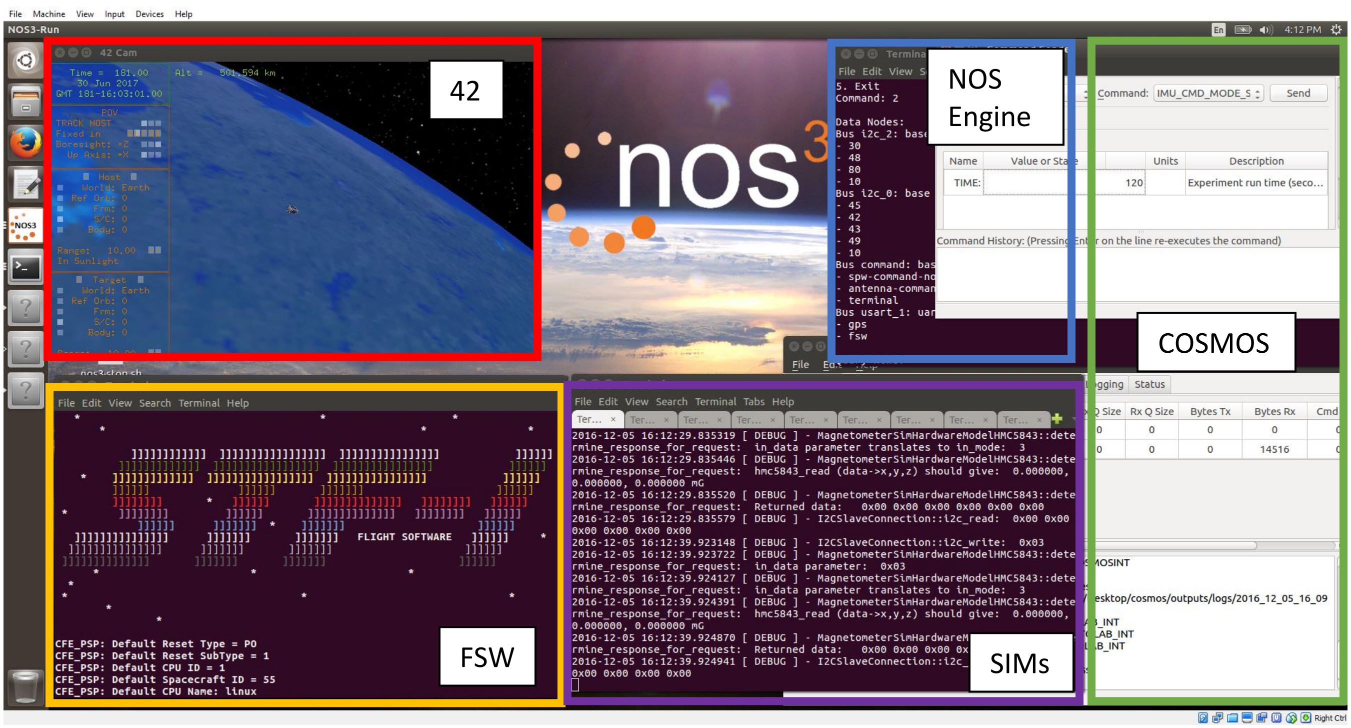


Figure 5 - NOS3 Running - Server, Simulators, 42, Flight Software, COSMOS

2. "*NOS3 Flight Software*" is the last component to start up.
3. Once flight software starts, telemetry can be commanded to be sent to COSMOS and telemetry can be viewed by:
    a. In the COSMOS "*Command Sender*" window:
        i. Select "*Target:*" to be "*NOS3*"
        ii. The only "*Command:*" is "*Enable Telemetry*", so it will automatically populate
        iii. Click "***Send***"

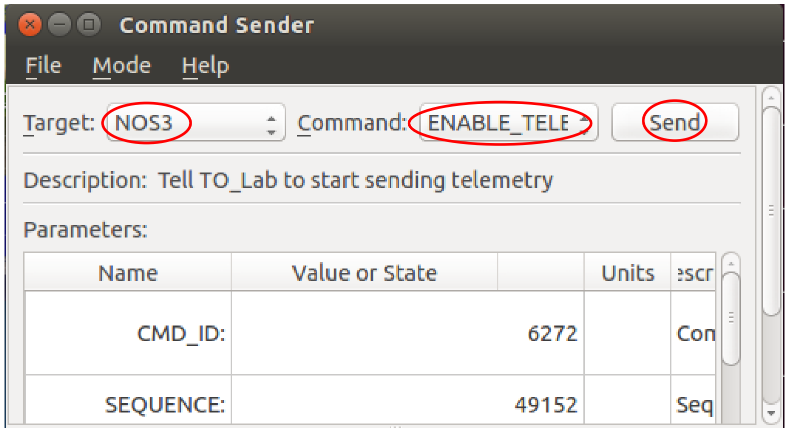


Figure 6 - COSMOS Command Sender - Enabling Telemetry

b. In the COSMOS "*Command and Telemetry Server – NOS3 Configuration*" window:
   i. "*Bytes Tx*" and "*Cmd Pkts*" should change from 0 to a positive number
   ii. "*Bytes Rx*" and "*Tlm Pkts*" should start counting up as telemetry is received

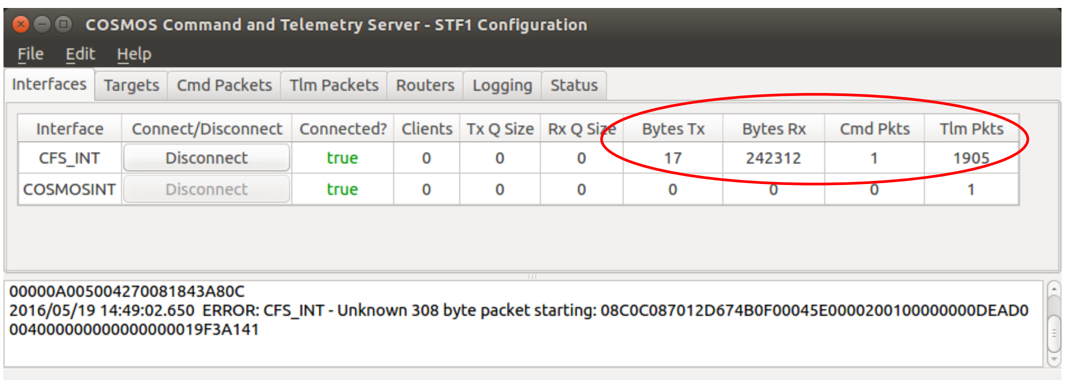


Figure 7 - COSMOS Command and Telemetry Server - Showing "Bytes Tx", "Bytes Rx"

c. In the COSMOS "*Command and Telemetry Server – NOS3 Configuration*" window:
   i. Click on the "*Tlm Packets*" tab
   ii. Scroll down to Target Name "*NOS3*" and Packet Name "*NOS3_NAV_MSG*"
   iii. Click on "***View in Packet Viewer***".

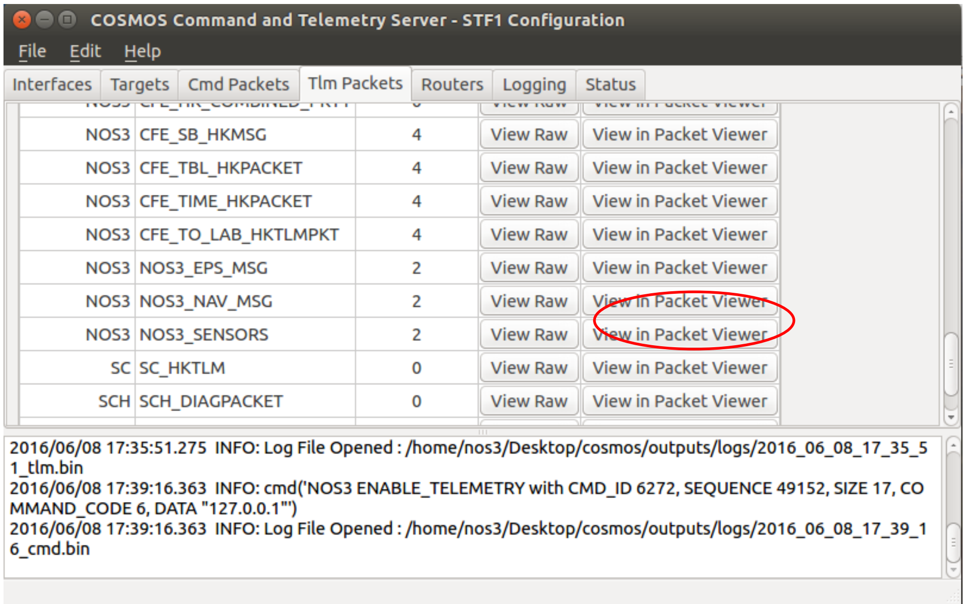


Figure 8 - COSMOS Command and Telemetry Server - Selecting "View in Packet Viewer" for "NOS3_NAV_MSG"

d. A "*Packet Viewer*" window will be displayed showing the formatted telemetry of the NOS3 navigation message containing data received by the flight software from the GPS simulator and packaged and sent down as telemetry. This navigation telemetry packet is configured to be sent approximately every 10 seconds.

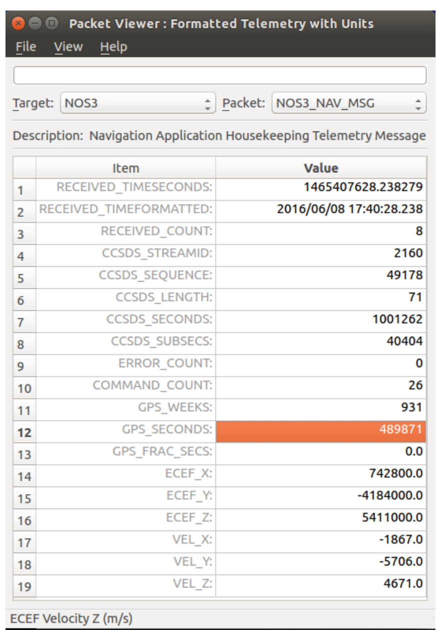


Figure 9 - COSMOS Packet Viewer – NOS3_NAV_MSG

4. To stop all NOS3 software, double click "***nos3-stop.sh***".

5 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 NOS^3 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 NOS^3 is to obtain the *nos3* code repository. Currently, that repository has a folder structure like the following:

1. nos3\
    a. apps\ - applications provided with the Core Flight System (cFS)
    b. cfe\ - the central Core Flight Executive (cFE) portion of cFS which provides essential services
    c. nanomind\ - header files particular to the Nanomind Single Board Computer in use by STF-1
    d. osal\ - the Operating System Abstraction Layer that is used with cFE/cFS to provide a common foundation for developing flight software
    e. psp\ - platform support package code that provides board specific code for various boards, CPUs, and operating systems
    f. sim\ - all of the software simulation code that simulates hardware components in NOS^3
    g. stf_apps\ - applications specific to STF-1
        i. Stf1ProjectSettings.cmake – Cmake settings for building STF-1 flight software and NOS^3 simulators
    h. stf_defs\ - specific configuration definition files for STF-1
    i. support\ - installers, planning tools, scripts, and other files needed for the NOS^3 virtual machine
        i. cfc\ - COSMOS File Creator tool and associated documentation
        ii. cosmos\ - command and telemetry configuration data to install in COSMOS
        iii. installers\ - scripts and necessary packages
        iv. planning\ - OIPP tool
        v. VirtualMachine\ - scripts to be moved to the desktop for easy access
        vi. Vagrantfile – The configuration file for vagrant that describes the NOS^3 Virtual Machine and how to configure and provision it
    j. tools\ - tools that accompany cFS
    k. CMakeLists.txt – top level CMake file for building STF-1 flight software and NOS^3 simulators
    l. README.txt – basic summary of the NOS^3 package and all the parts

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.

To elaborate on the quick start guide, to create the NOS^3 virtual machine the steps are:

1. Open a command terminal and navigate to directory where NOS^3 was unzipped

```
Command Prompt
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\JPL>cd NOS3_v1.02.00

C:\Users\JPL\NOS3_v1.02.00>
```

Figure 10 - Command Terminal

2. Execute "*windows_nos3_installer.bat*", "*linux_nos3_installer.sh*", "*pi_nos3_installer.sh*".
    a. Note if using linux or pi, may need to dos2unix the file before running. Example: 'dos2unix linux_nos3_installer.sh'

```
Command Prompt
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\JPL>cd NOS3_v1.02.00

C:\Users\JPL\NOS3_v1.02.00>windows_nos3_installer.bat
```

Figure 11 - Run Installer Script

3. The command prompt will be renamed with the disclaimer displayed

```
NASA Operational Simulator for Small Satellites (NOS3)

Welcome to the Nasa Operational Simulator for Small Satellites (NOS3) Installer!

----------------
-- Disclaimer --
----------------
This software is provided ''as is'' without any warranty of any, kind either express, implied
, or statutory, including, but not limited to, any warranty that the software will conform to
, specifications any implied warranties of merchantability, fitness for a particular purpose,
 and freedom from infringement, and any warranty that the documentation will conform to the p
rogram, or any warranty that the software will be error free.  In no event shall NASA be liab
le for any damages, including, but not limited to direct, indirect, special or consequential
damages, arising out of, resulting from, or in any way connected with the software or its doc
umentation.  Whether or not based upon warranty, contract, tort or otherwise, and whether or
not loss was sustained from, or arose out of the results of, or use of, the software, documen
tation or services provided hereunder.

Press 'Y' to confirm you have read the disclaimer...
[Y,N]?
```

Figure 12 – Disclaimer

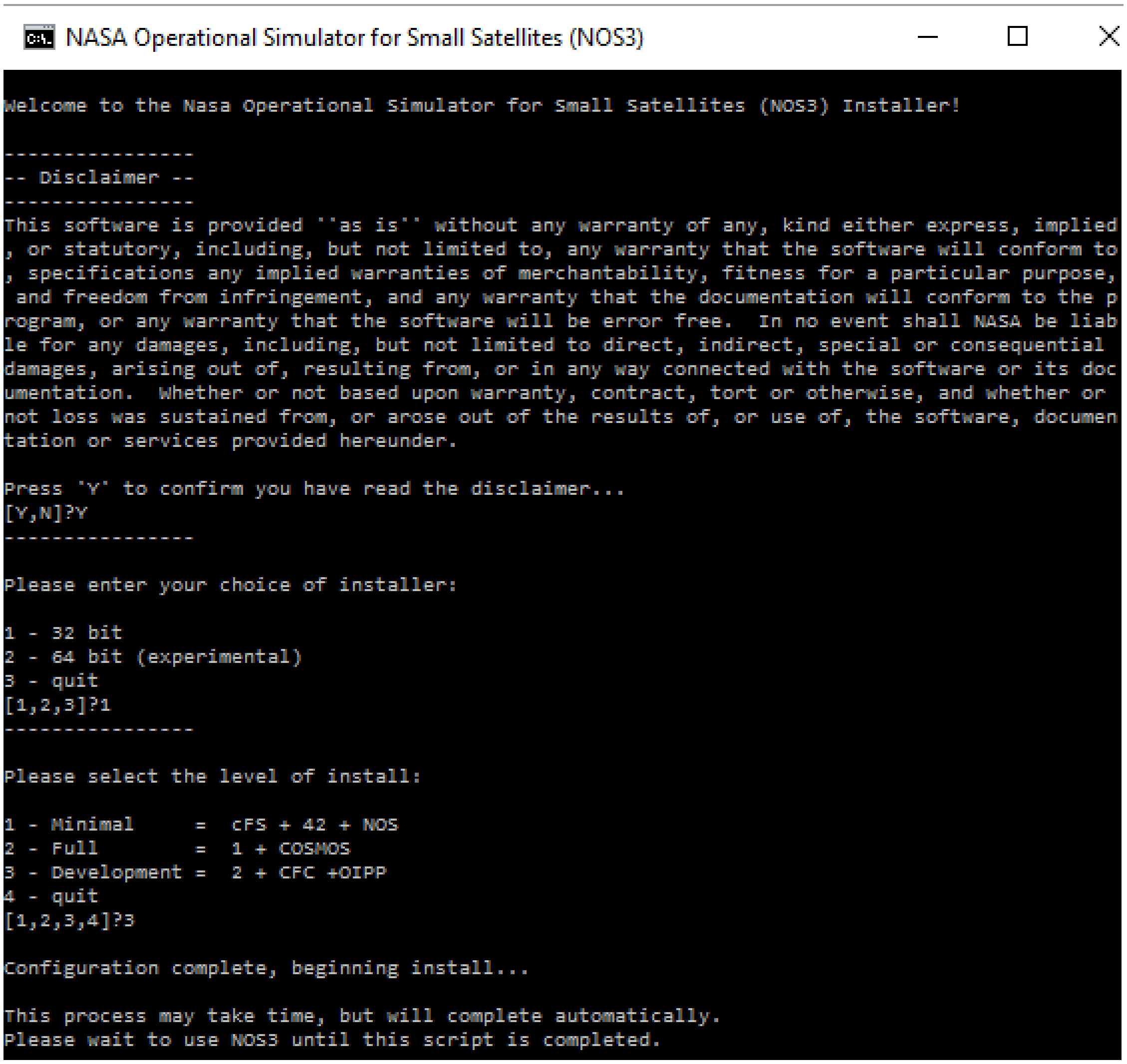
4. Confirm you have read the disclaimer by pressing 'Y'

```
Please select the level of install:

1 - Minimal     =  cFS + 42 + NOS
2 - Full        =  1 + COSMOS
3 - Development =  2 + CFC +OIPP
4 - quit
[1,2,3,4]?3
```

Figure 13 - Level of Install

5. The level of install by entering 1, 2, or 3
    a. Each level builds upon the previous one with minimal, or ‘1’, being the base



```
NASA Operational Simulator for Small Satellites (NOS3)

Welcome to the Nasa Operational Simulator for Small Satellites (NOS3) Installer!

----------------
-- Disclaimer --
----------------
This software is provided ''as is'' without any warranty of any, kind either express, implied
, or statutory, including, but not limited to, any warranty that the software will conform to
, specifications any implied warranties of merchantability, fitness for a particular purpose,
 and freedom from infringement, and any warranty that the documentation will conform to the p
rogram, or any warranty that the software will be error free.  In no event shall NASA be liab
le for any damages, including, but not limited to direct, indirect, special or consequential
damages, arising out of, resulting from, or in any way connected with the software or its doc
umentation.  Whether or not based upon warranty, contract, tort or otherwise, and whether or
not loss was sustained from, or arose out of the results of, or use of, the software, documen
tation or services provided hereunder.

Press 'Y' to confirm you have read the disclaimer...
[Y,N]?Y
----------------

Please enter your choice of installer:

1 - 32 bit
2 - 64 bit (experimental)
3 - quit
[1,2,3]?1
----------------

Please select the level of install:

1 - Minimal     =  cFS + 42 + NOS
2 - Full        =  1 + COSMOS
3 - Development =  2 + CFC +OIPP
4 - quit
[1,2,3,4]?3

Configuration complete, beginning install...

This process may take time, but will complete automatically.
Please wait to use NOS3 until this script is completed.
```

Figure 14 - Complete Configuration

6. Wait for the installation process to complete
    a. At several points the script may seem to hang, but is suppressed due to excessive output
    b. Completion is certain once the script exits as displayed below

NASA Operational Simulator for Small Satellites (NOS3)

```
==> default: --- Cleanup ---
==> default:
==> default: Reading package lists...
==> default: Building dependency tree...
==> default: Reading state information...
==> default: 0 upgraded, 0 newly installed, 0 to remove and 6 not upgraded.
==> default: success!
==> default:
==> default: --- Exiting shell provisioner ---
==> default:
==> default: Configuring cache buckets...
==> default: Running provisioner: shell...
    default: Running: inline script
==> default: stdin: is not a tty
==> default: ***** Run `vagrant reload` to start the graphical environment if necessary *****

==> default: Configuring cache buckets...
==> default: Attempting graceful shutdown of VM...
==> default: Checking if box 'ubuntu/trusty32' is up to date...
==> default: A newer version of the box 'ubuntu/trusty32' is available! You currently
==> default: have version '20160627.0.1'. The latest is version '20160720.0.0'. Run
==> default: `vagrant box update` to update.
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
    default: The guest additions on this VM do not match the installed version of
    default: VirtualBox! In most cases this is fine, but in rare cases it can
    default: prevent things such as shared folders from working properly. If you see
    default: shared folder errors, please make sure the guest additions within the
    default: virtual machine match the version of VirtualBox you have installed on
    default: your host and reload your VM.
    default:
    default: Guest Additions Version: 4.3.36
    default: VirtualBox Version: 5.0
==> default: Mounting shared folders...
    default: /vagrant => C:/Users/JPL/NOS3_v1.02.00/nos3/support
    default: /vagrant_parent => C:/Users/JPL/NOS3_v1.02.00/nos3
    default: /tmp/vagrant-cache => C:/Users/JPL/.vagrant.d/cache/ubuntu/trusty32
==> default: Configuring cache buckets...
==> default: Machine already provisioned. Run `vagrant provision` or use the `--provision`
==> default: flag to force provisioning. Provisioners marked to run always will still run.

Exiting the NOS3 installer now! NOS3 is ready for use!

C:\Users\JPL\NOS3_v1.02.00\nos3\support>
```

Figure 15 - Installation Complete

7. If the screen has locked, log in using:
    a. Username: nos3
    b. Password: nos3123!

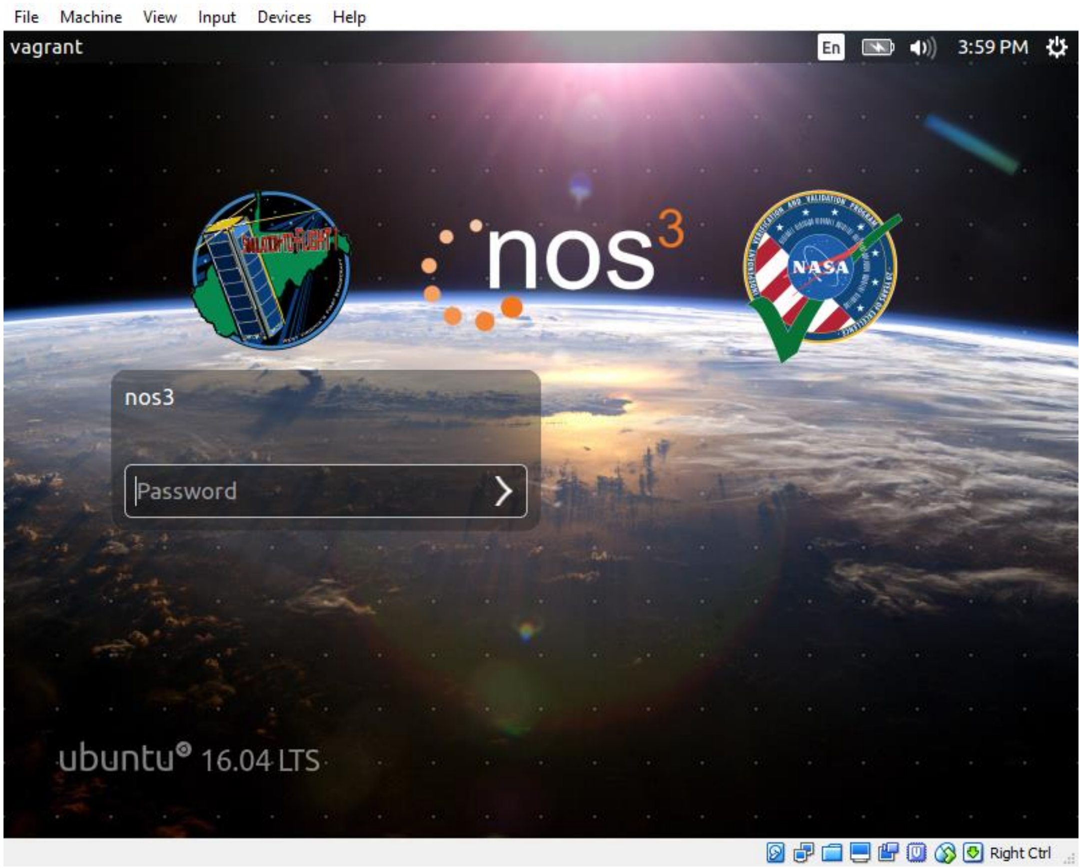


Figure 16 - Lock Screen

8. The initial desktop seen upon login should mimic that displayed below

Figure 17 - Initial Desktop

This concludes the initial installation and log in on the NOS^3 virtual machine.

5.1 The Vagrantfile and Process

The following section describes the provisioning that is done in the NOS^3 Vagrantfile using Vagrant. Vagrantfiles are text files written in a language called Ruby.

5.1.1 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.

5.1.2 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 NOS^3, this base box is a very minimal installation of Ubuntu Linux, Version 14 (Trusty). 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.

5.1.3 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 NOS^3. 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 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.

5.1.4 Conclusion and References

This concludes an overview of the Vagrantfile which is used to install and configure the NOS^3 environment. For more details, please consult the Vagrantfile itself.

5.2 Host Installation

Utilizing the provisioning scripts mentioned above, host installation on Ubuntu 16.04 is possible. These files can be found at 'nos3/support/installers/' and are named 'nos3_64_*.sh'. These scripts require the specific package names found for Ubuntu 16.04 in order to properly execute. These provisioner scripts are broken into minimum, full, and developer levels as mentioned above. Install in order to the level desired. It is recommended that these scripts be thoughly reviewed prior to running on the host as a lot of file installation and manipulation is performed.

6 Building NOS^3 Components: Flight Software and Simulators

6.1 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!

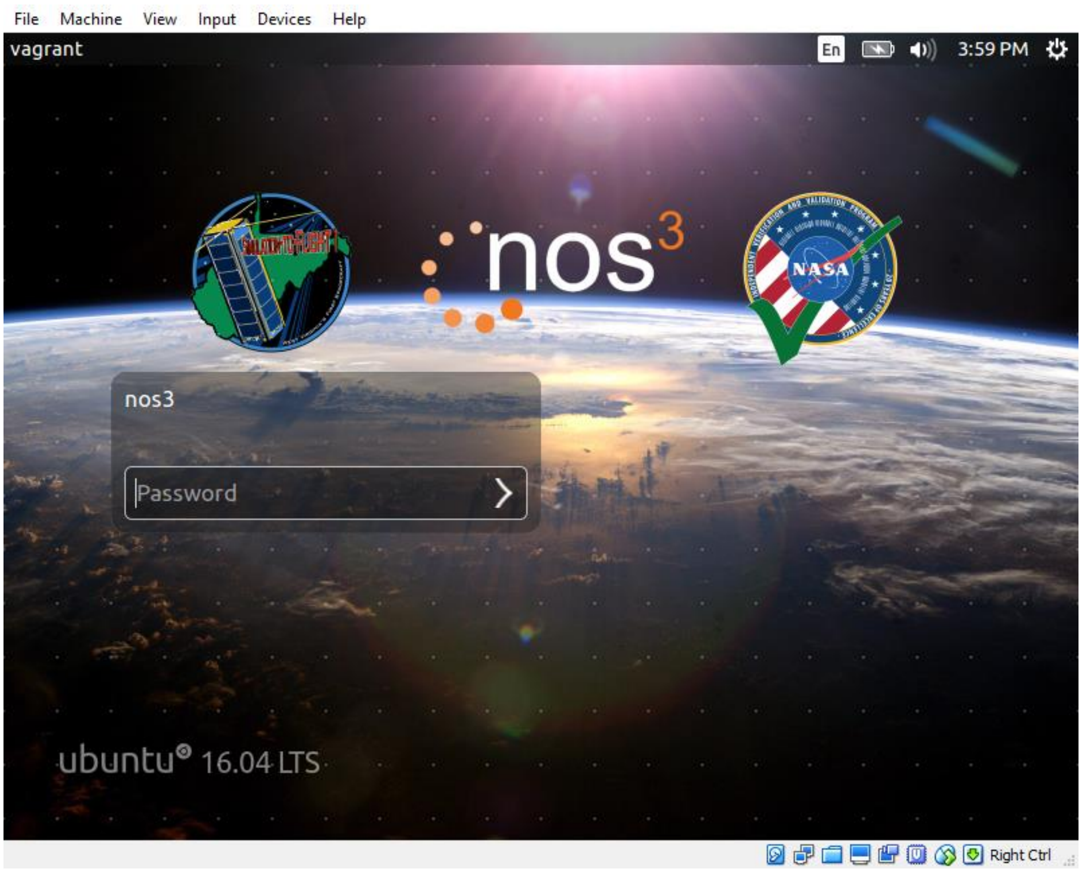


Figure 18 - Ubuntu Linux Desktop Greeter

3. Double click “***nos3-build.sh***”

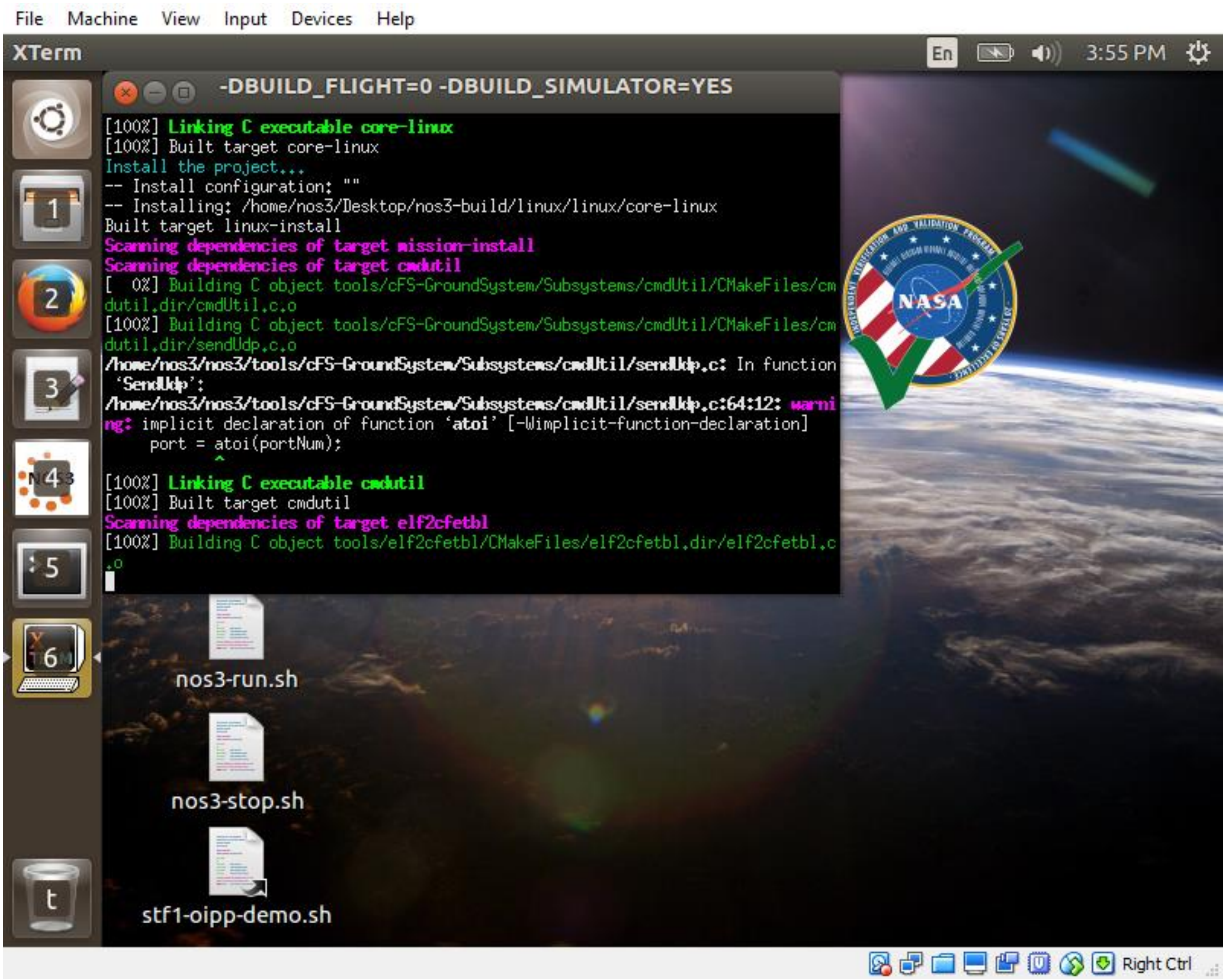


Figure 19 - Double Click "first-nos3-build.sh"

4. A terminal window will come up and build the simulator and flight software (this will take quite a few minutes)

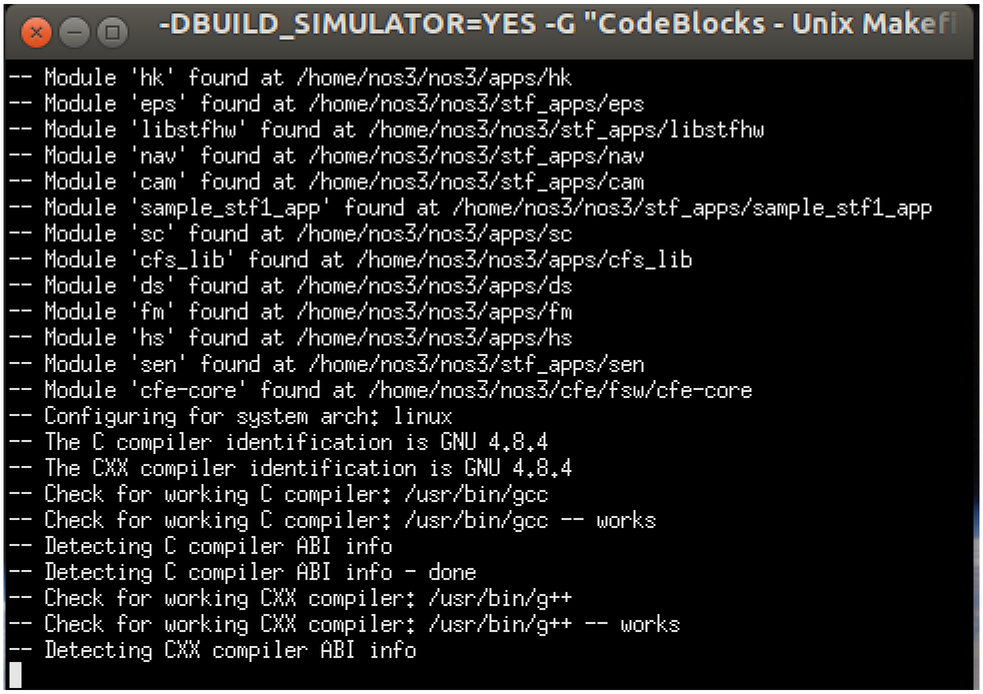


Figure 20 – NOS3 Building: CMake Execution

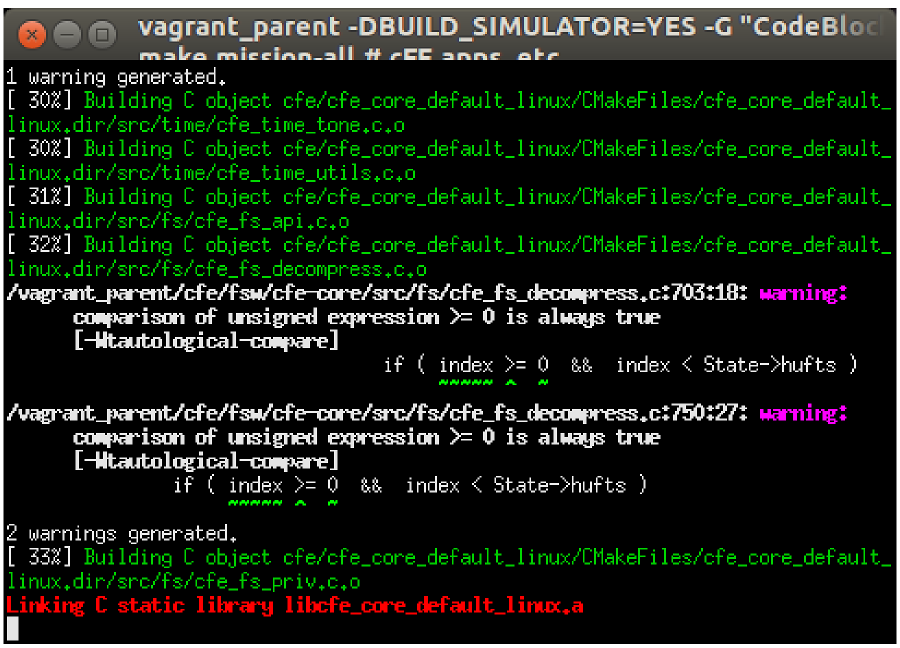


Figure 21 – NOS3 Building - Make and Install the Flight Software

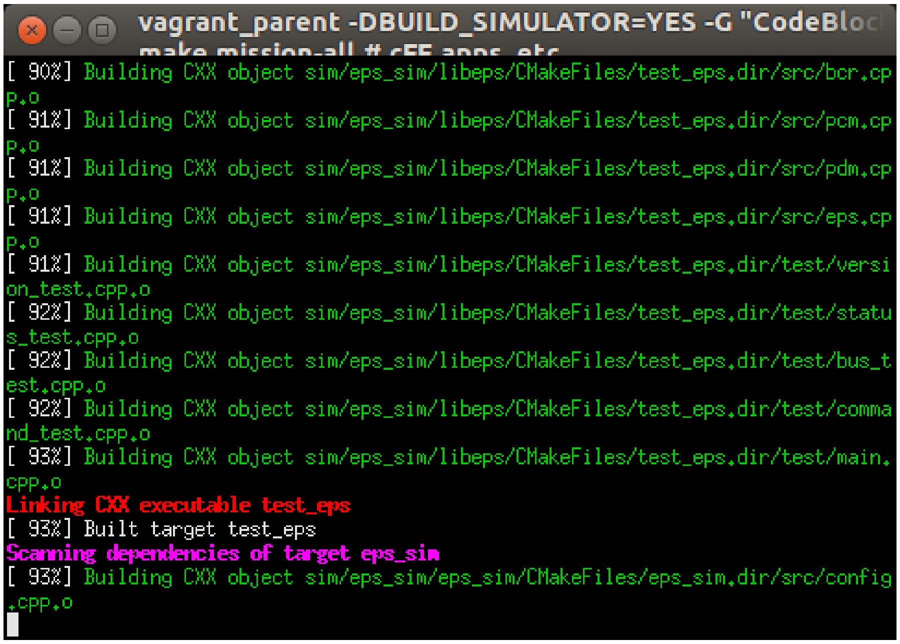


Figure 22 – NOS3 Building - Make and Install Simulators

5. When this process is complete, a new folder "*nos3-build*" should exist on the desktop with the built software.

6.2 The Build Script

Inside the build script, the following items are being executed:

1. *cmake ~/nos3 -DBUILD_SIMULATOR=YES -G "CodeBlocks - Unix Makefiles"*

   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 CMake runs, it creates makefiles and other build files necessary for compiling, linking, and installing the flight software and simulators. These files are created in the directory "*/home/nos3/Desktop/nos3-build*".

2. *make mission-all* and *make mission-install*

   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 mission-all*" is run, the flight software is compiled and linked in subdirectories of the directory "*/home/nos3/Desktop/nos3-build*". When "*make mission-install*" is run, the flight software is installed in the directory "*/home/nos3/Desktop/nos3-build/linux/linux*". This includes the "*core-linux*" executable and "*cf*" directory which contains the shared objects, libraries, and configuration tables and files for the flight software.

3. *make* and *make install*

   When "*make*" is run, the simulators are compiled and linked in subdirectories of the directory *"/home/nos3/Desktop/nos3-build"*. When "*make install*" is run, the simulation software executables and configuration files are installed in the directory "*/home/nos3/Desktop/nos3-build/bin*" and the simulation software libraries are installed in the directory "*/home/nos3/Desktop/nos3-build/lib*".

7 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!

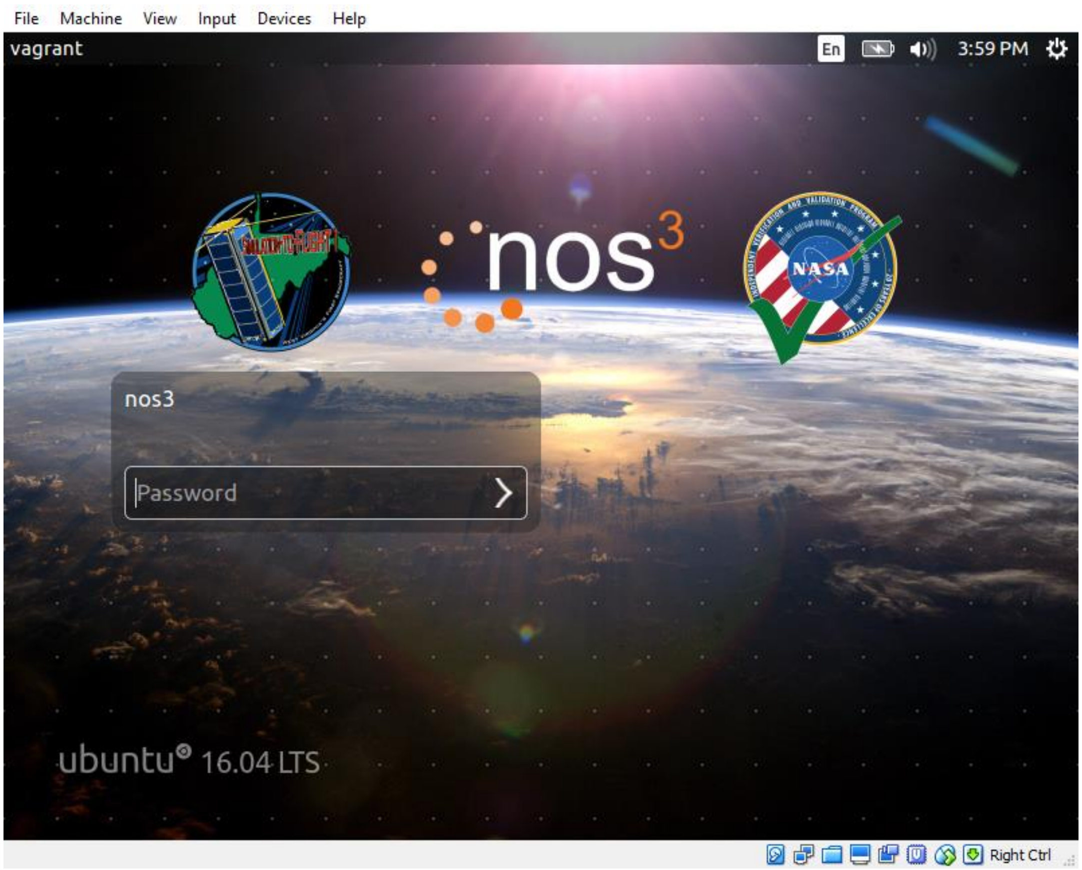


Figure 23 - Ubuntu Linux Desktop Greeter

3. Double click "***nos3-run.sh***".

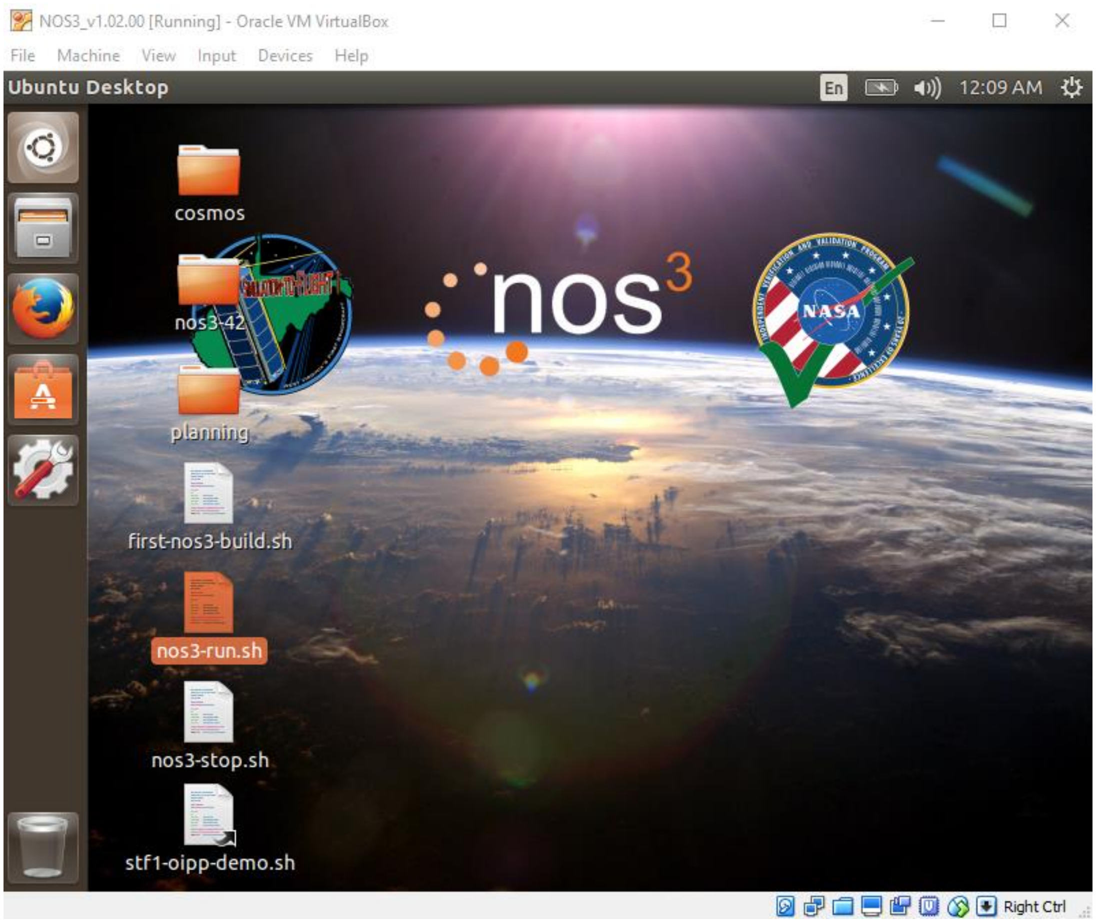


Figure 24 - Double Click "nos3-run.sh"

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

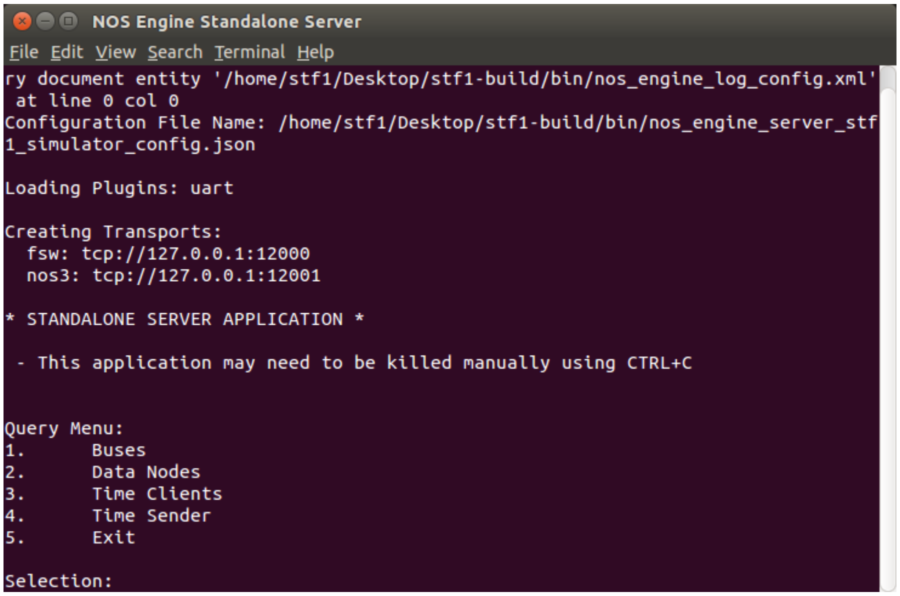


Figure 25 - NOS Engine Standalone Server

The NOS Engine Standalone Server provides the software simulated communication bus structure that is used by NOS^3 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 NOS^3, the server is configured using the file */home/nos3/Desktop/nos3-build/bin/nos_engine_server_simulator_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)

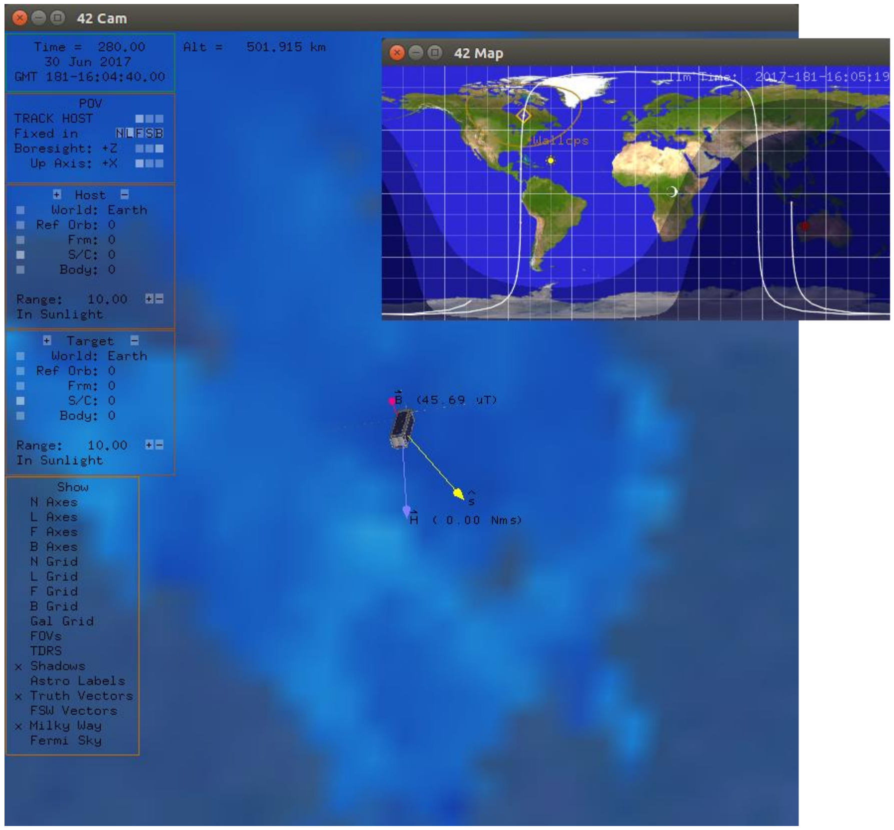


Figure 26 - 42 Dynamic Simulator

42 is a general-purpose, multi-body, multi-spacecraft simulation. For NOS^3, 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 NOS^3. 42 is open source C code. For NOS^3 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 */home/nos3/Desktop/nos3-42/NOS3-42InOut*. 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. STF1-TLE.txt – A two line element set for STF-1 referred to by Orb_LEO.txt. A two-line element set (TLE) is a data format encoding a list of orbital elements of an Earth-orbiting object for a given point in time, the epoch. Using suitable prediction formula, the state (position and velocity) at any point in the past or future can be estimated to some accuracy. The TLE data representation is specific to the simplified perturbations models (SGP, SGP4, SDP4, SGP8 and SDP8), so any algorithm using a TLE as a data source must implement one of the SGP models

to correctly compute the state at a time of interest. Prior to STF-1 launch the orbital elements are notional, based on a probable STF-1 orbit.

4. SC_STF1.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.
5. Inp_IPC.txt – File defining the TCP/IP or file parameters for communicating input and output to and from 42.
6. 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.
7. There are several other input files which are not used much for NOS^3, 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. COSMOS (GUI windows for Legal Agreement, COSMOS Command and Telemetry Server – STF1 Configuration, Command Sender, Launcher)

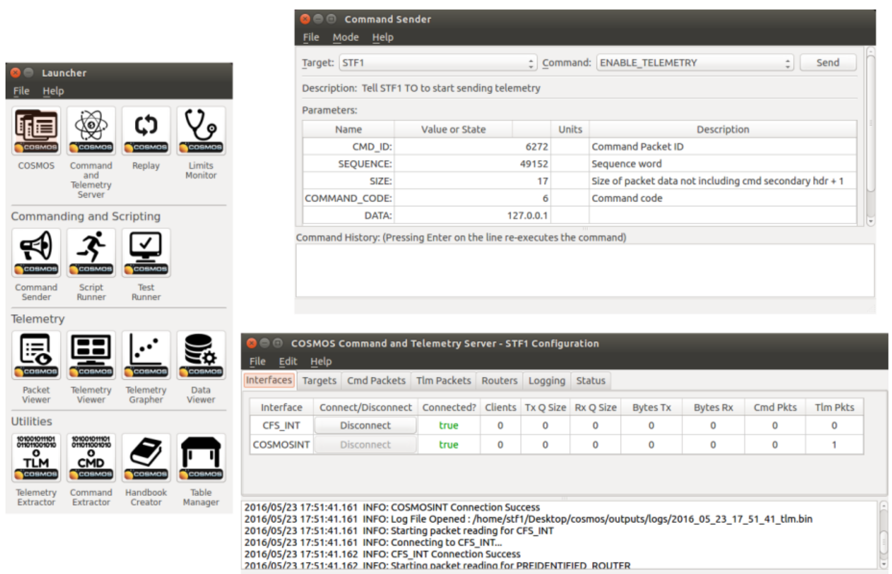


Figure 27 – COSMOS

COSMOS is stated to be "The User Interface for Command and Control of Embedded Systems". It is used by NOS^3 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 executing "*cosmos install cosmos*" on the NOS3 user's desktop and then copying configuration files which define the NOS3 commands and telemetry into subdirectories of */home/nos3/Desktop/cosmos*.

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

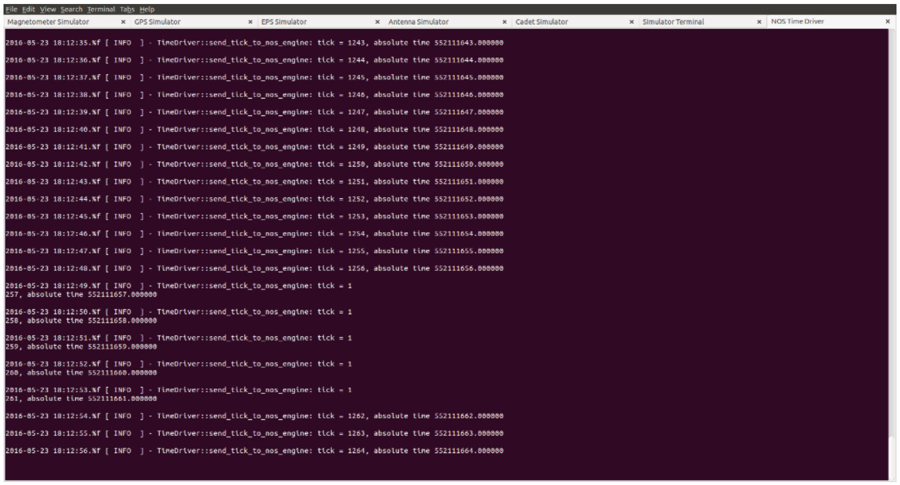


Figure 28 – Simulators

Currently, NOS^3 starts 7 simulators:

1. Magnetometer Simulator – Simulates a hardware magnetometer using magnetic field environmental data from 42.
2. GPS Simulator – Simulates a hardware GPS using position and velocity data from 42.
3. EPS Simulator – Simulates a hardware electrical power system.
4. Antenna Simulator – Simulates an antenna found on a cubesat including the antenna deployment mechanisms.
5. Cadet Simulator – Simulates a UHF radio found on a cubesat.
6. 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.
7. 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 */home/nos3/Desktop/nos3-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)

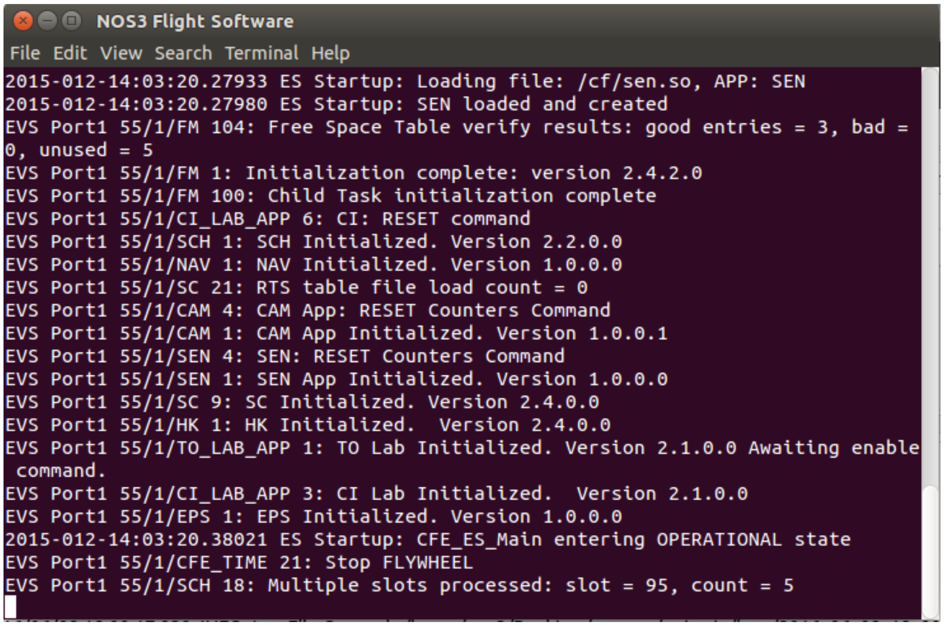


Figure 29 – NOS3 Flight Software

Last, but certainly not least is the NOS^3 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.

8 NOS^3 Workflows

Two workflows are currently known to utilize NOS^3 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…

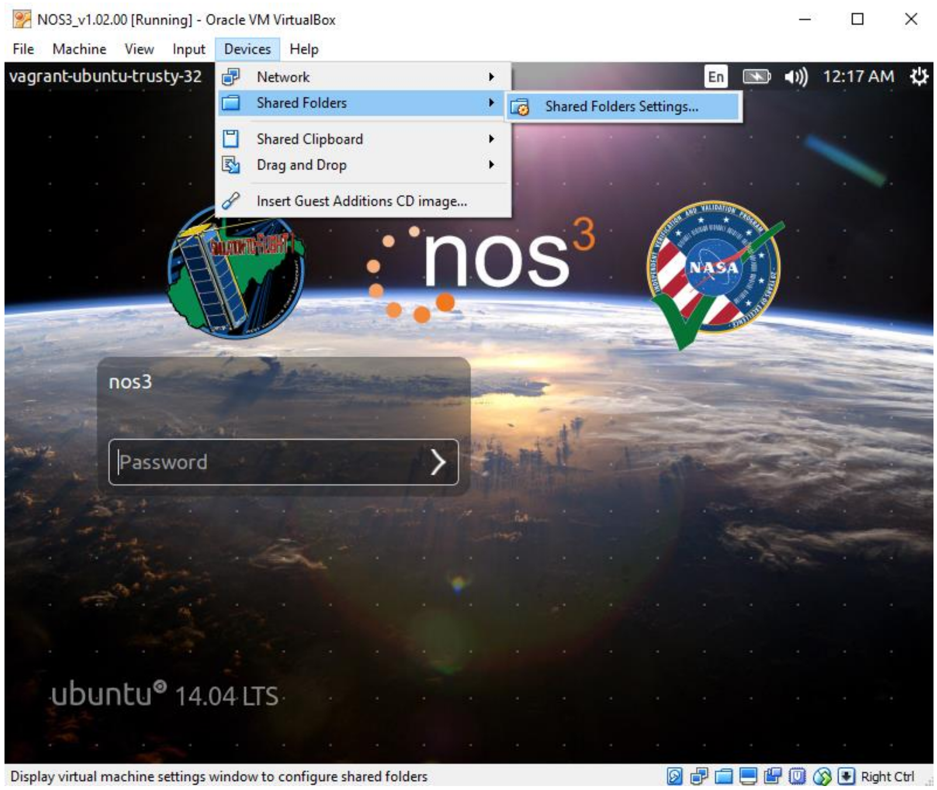


Figure 30 - Shared Folder Settings

2. Add the unzipped 'nos3' folder to the list of shared folders

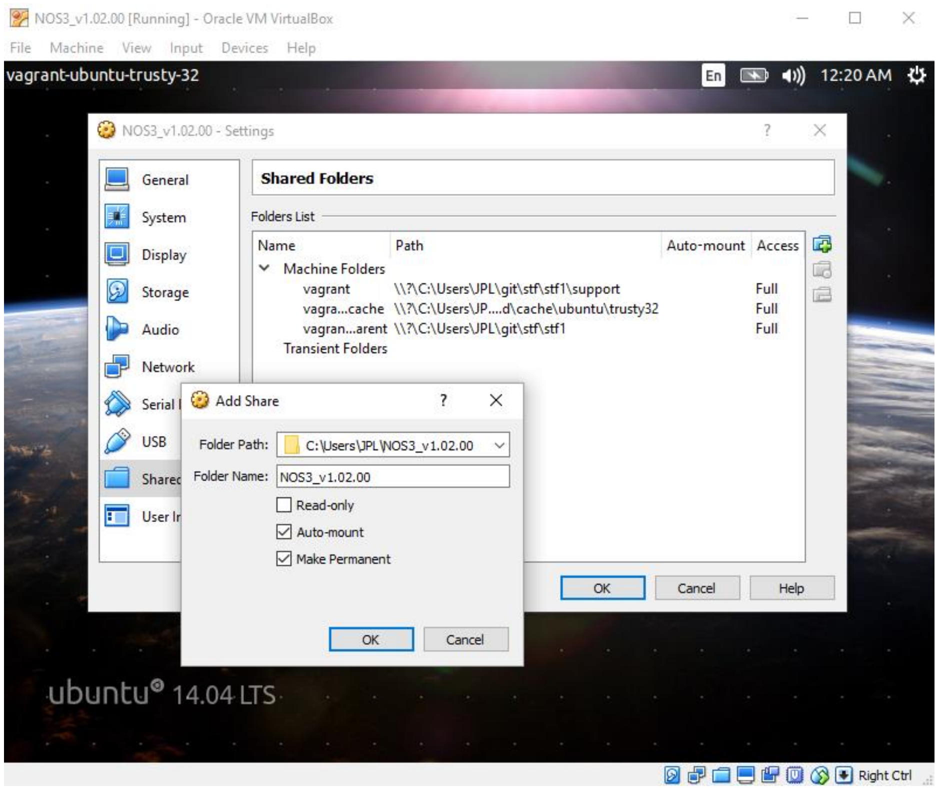


Figure 31 - Share in NOS3 Folder

3. Archive the current 'nos3' folder in the VM
   a. In a terminal enter the following command: 'sudo mv nos3/* nos3_old/'
4. Mount the newly shared one
   a. In a terminal enter the following command: 'sudo mount –t vboxsf ~/nos3 ~/nos3'

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

9 Hardware Simulator Framework / Example Simulator

NOS^3 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), 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).

9.1 Background and Supporting Concepts

9.1.1 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 NOS^3 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 NOS^3 simulator code base. Instead of the shapes and shape factory in the article, the components in NOS^3 simulators which are constructed via factories are hardware models and data providers.

9.1.2 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.

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

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

9.2.1 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.absolute-start-time` – The absolute start time of the simulation in decimal seconds from the J2000 epoch.
3. `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;
   ```
4. `simulator.name` – The name you gave your simulator; it should agree with the string you put in the `main` function (see below).
5. `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).
6. `simulator.hardware-model.type` – The name string for your hardware model.
7. `simulator.hardware-model.connections` – A list of <connection></connection> tags which describes the connections that the hardware model has.
8. `simulator.hardware-model.data-provider` – Information on the data provider (if one is used and created using the data provider factory).
9. `simulator.hardware-model.data-provider.type` – The name string for your data provider (if one is used).

9.2.2 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);
```

9.2.3 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");
```

9.2.4 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.

9.2.4.1 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 NOS^3. 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.

9.2.4.2 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();`

9.2.4.3 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 constructor, make the call:
`_uart_connection->close();`

9.3 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>fpp1</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).

9.4 Example Simulator

Hopefully this introduction is useful in describing the flexible, extensible framework employed in developing NOS^3 simulators. This introduction has attempted to describe the design pattern used within NOS^3 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 NOS^3 simulation environment.

For a complete example, refer to the source code and CMakeLists.txt file in the `nos3` git repository, subdirectory `sim/example_sim/` and refer to the configuration file in the `nos3` git repository, file `sim/sim_common/cfg/nos3-simulator.xml` (see the simulator section with name “`example`”). Note also that if a new simulator’s `CMakeLists.txt` file for a simulator has a project name line like “`project(example_sim)`” at the, the line “`add_subdirectory(example_sim)`” must be added to the bottom of the `sim/CMakeLists.txt` file in the `nos3` git repository so that the new simulator will be built.

10 42, A Visualization and Simulation Tool for Spacecraft Orbit and Attitude Dynamics

10.1 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 NOS^3 (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 NOS^3 (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 NOS^3, the main configuration files of interest are the following:

8. 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.
9. 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.
10. STF1-TLE.txt – A two line element set for STF-1 referred to by Orb_LEO.txt. A two-line element set (TLE) is a data format encoding a list of orbital elements of an Earth-orbiting object for a given point in time, the epoch. Using suitable prediction formula, the state (position and velocity) at any point in the past or future can be estimated to some accuracy. The TLE data representation is specific to the simplified perturbations models (SGP, SGP4, SDP4, SGP8 and SDP8), so any algorithm using a TLE as a data source must implement one of the SGP models to correctly compute the state at a time of interest. Prior to STF-1 launch the orbital elements are notional, based on a probable STF-1 orbit.

11. SC_STF1.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.
12. Inp_IPC.txt – File defining the TCP/IP or file parameters for communicating input and output to and from 42.
13. 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.
14. There are several other input files which are not used much for NOS^3, 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).

10.2 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 NOS^3 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.

10.3 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" file is modified as follows. Change the "Time Mode" line to have the value "SIMULATION". Set the "Sim Time Connection String" line to have the connection string for contacting the NOS Engine Standalone Server. Set the "Sim Time Bus" line to the NOS Engine bus name to use to retrieve time from.

10.4 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.

11 Flight Software Development, Especially Using cFS

The preferred operating system for use with NOS^3 is the open-source Core Flight System (cFS) originally developed by NASA GSFC. This section will describe the method utilized to interface NOS^3 with cFS, as well as a generic method to interface with any flight software that can compile for Linux.

11.1 cFS and NOS^3

11.1.1 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 NOS^3 an ideal development environment when using the OSAL Linux target.

6.1.2 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 NOS^3 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 NOS^3 components are running.

6.1.3 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 NOS^3 driver source code and the executable can be run in the NOS^3 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 NOS^3 source.

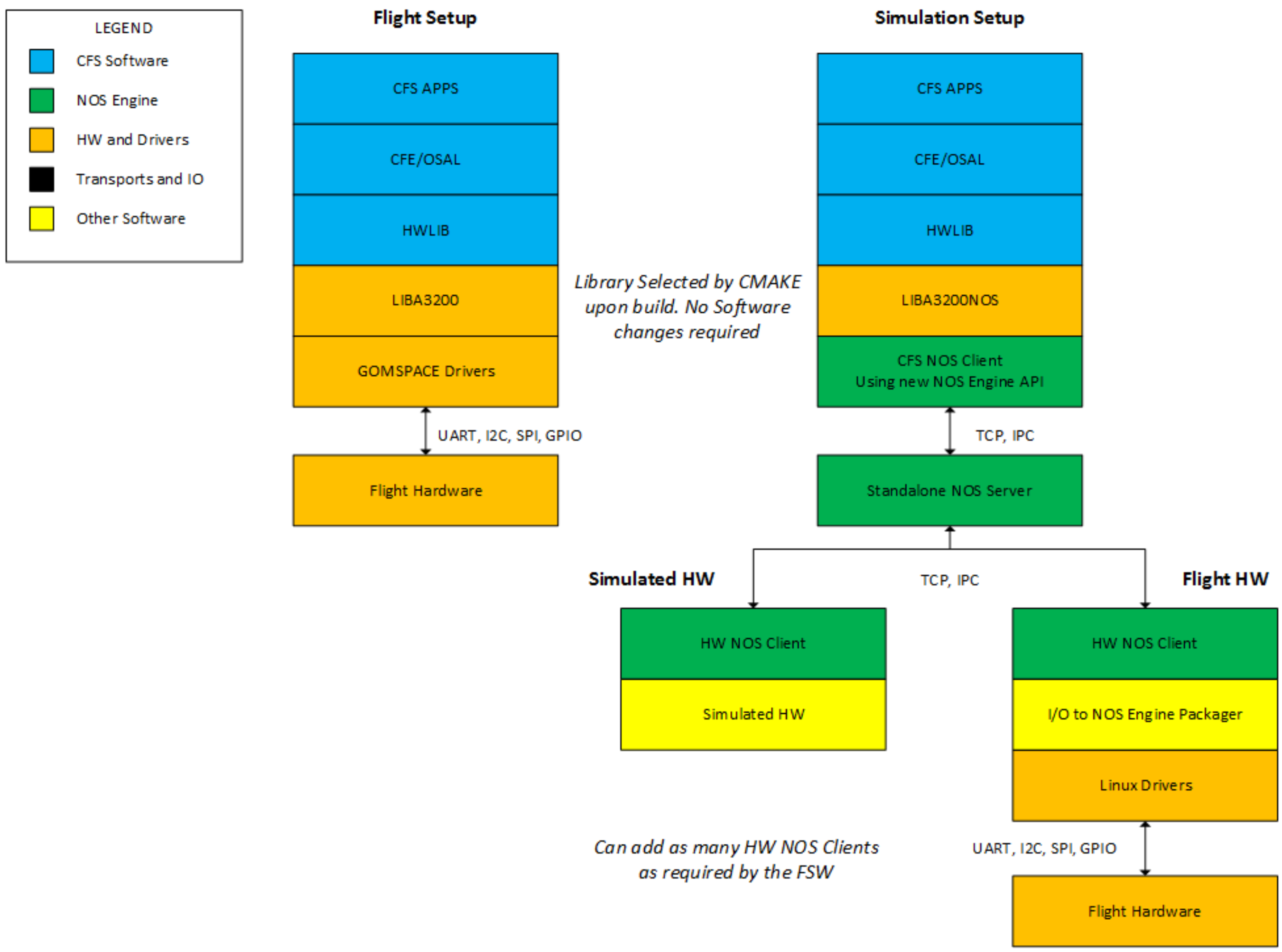


Figure 32 - Flight and Simulation Targets

6.2 Connecting cFS to NOS^3

In order to use NOS^3 with cFS, modifications are required to the open-source release. The recommended method for using NOS^3 is described in the NOS^3 User's Guide, in which these modifications have already been made. If not using the cFS included with the NOS^3 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 *proj/proj_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. Copy the *toolchain-linux.cmake* from the *nos3/stf1_defs* directory into the *proj/proj_defs* directory.
3. Copy the *nos-linux* PSP from the *nos3/psp/fsw* directory into the *proj/psp/fsw* directory.
4. Add a *proj/proj_apps* directory and copy apps from *nos3/stf_apps* as needed.

5. Create a *proj/proj_apps/libhw* or copy the *nos3/stf_apps/libstfhw* directory.
    a. The CMakeLists.txt file in *nos3/stf_apps/libstfhw* 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 NOS^3 drivers for I/O. (See section 6.3 for an example driver)
6. Copy the needed simulation components from the *nos3/sim* directory to *proj/sim*.

6.3 NOS^3 Drivers and Other FSW

It is possible to connect NOS^3 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 NOS^3 drivers as described in previous sections.

6.3.1 Writing a NOS^3 Driver

The NOS^3 source is the best resource for examples to aid in writing a new NOS^3 driver. The GPS library, GomSpace 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, but this application could just as easily be any other FSW source file.

6.3.1.1 Application and Hardware Library

The application using 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 GPS 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 *GPS_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/stf_apps/nav/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 */
GPS_ReadAvailableData(DataBuffer, &DataLen);

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

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

```
#include <dev/usart.h>

/* some code removed for readability see nos3/stf_apps/libstfhw/fsw/src/gps_lib.c */

/*
** Called by any cFS app that wants GPS data.
*/
void GPS_ReadAvailableData(uint8_t *DataBuffer, int32 *DataLen)
{
            int32 i = 0;

            /* TODO does this need to be sent periodically? */
            char gps_cmd[] = "log bestxyza";
            usart_putstr(GPS_UART, gps_cmd, strlen(gps_cmd));

            /* check how many bytes are waiting on the uart */
            *DataLen = usart_messages_waiting(GPS_UART);

            /* declare an out buffer to hold that data */
            if (*DataLen > 0)
            {
                /* grab a byte at a time from the uart and place into the buffer */
                while (i < (*DataLen))
                {
                    DataBuffer[i] = usart_getc(GPS_UART);
                    ++i;
                }
            }
            else
            {
                /* OS_printf("GPS_ReadAvailableData(): gps uart data len is 0\n"); */
            }
}
```

6.3.1.2 The NOS³ Driver

The example described above uses the usart.h header provided with the device drivers for the NanoMind A3200 being used by STF-1. This header is included by any library making calls to the USART and can be stored at any location. In this case the file is located at *nos3/nanomind/lib/libasf/gomspace/drivers/include/dev/usart.h.*

The functions used by the GPS library in this example are *usart_putstr*, *usart_messages_waiting*, and *usart_getc*, all of which are defined in *nos3/nanomind/lib/libasf/gomspace/drivers/avr32/usart.c.* The *usart_getc* function will be examined in more detail for this example. The code excerpt below shows the Nanomind function.

```
/**
 * Return next char in queue
 */
char usart_getc(int handle) {
        if (usart[handle] == NULL)
                return 0;
        char c;
        xQueueReceive(usart[handle]->usart_rxqueue, &c, portMAX_DELAY);
        return c;
}
```

The function above is used to return the next character from the USART buffer to the calling function. In order to write a driver for NOS³ this functionality must be mimicked in a new file with the same filename as the original drivers. The NOS³ USART driver for STF-1 is located at *nos3/stf_apps/libstfhw/sim/libasfnos/gomspace/drivers/avr32/usart,* and the usart_getc function defined in this file is shown in the code exerpt below.

```
char usart_getc(int handle)
{
    char c = 0;
    Uart *uart = get_usart_device(handle);
    if(uart)
    {
        /* TODO check return code */
        uart_getc(uart, (uint8_t*)&c);
    }
    return c;
}
```

The uart_getc functions 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. Typically all functions from the OBC driver would be implemented in the NOS^3 driver, however, only those called by the hardware library are necessary.

6.3.1.3 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 NOS^3 and can accomplish this swap easily. As described in section 6.2 the *CMakeLists.txt* file in *nos3/stf_apps/libstfhw* will provide the best example of how to include driver source code. A code exerpt from the CMAKE file can be seen below. The *CFE_SYSTEM_PSPNAME* is set in the toolchain file located at *nos3/stf1_defs/toolchain-linux.cmake*, and the call below checks the status of this *PSPNAME* and includes NOS^3 include paths and drivers as needed. In FSW that is not cFS, this IF statement could be edited to check a CMAKE defined value.

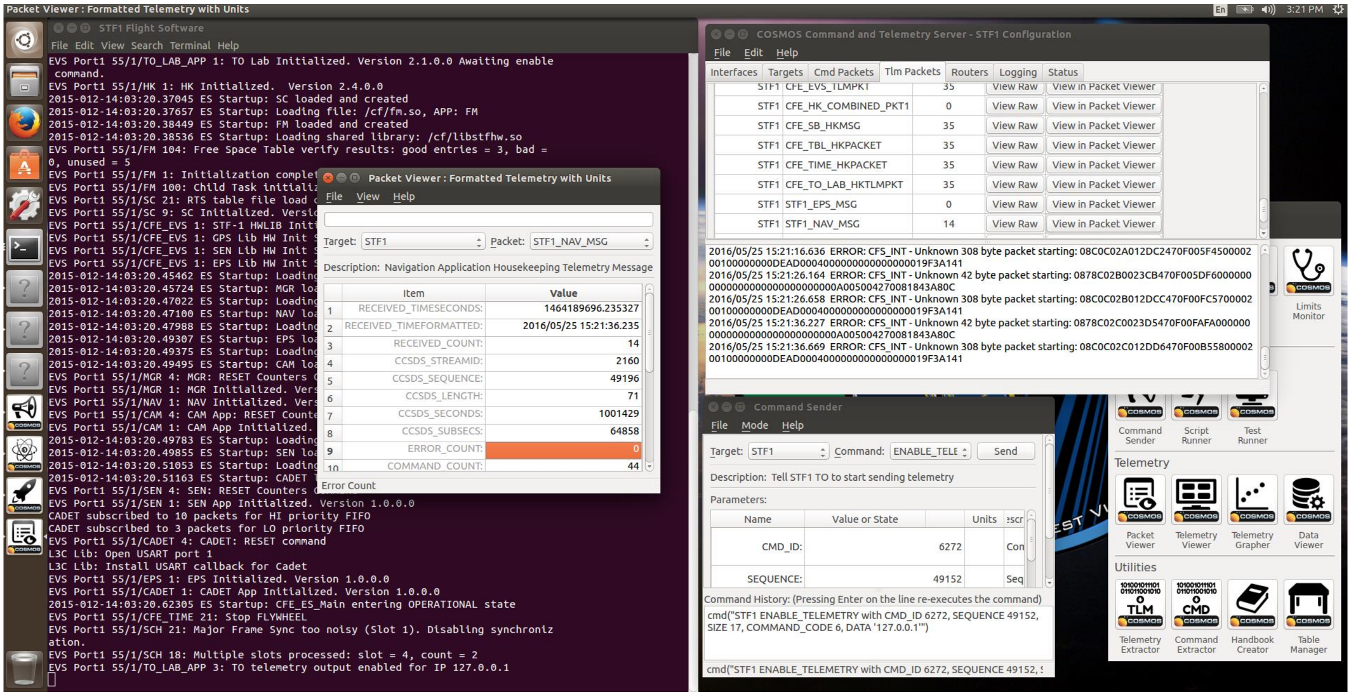
```
IF (CFE_SYSTEM_PSPNAME STREQUAL nos-linux)
    add_subdirectory(sim)
    include_directories(${nos-engine-link_SOURCE_DIR}/include)
    message(STATUS "Set NOS Include Directories")
ELSE ()
END IF()

/* some code removed for readability see nos3/stf_apps/libstfhw/CMakeLists.txt */

IF (CFE_SYSTEM_PSPNAME STREQUAL nos-linux)
    set_target_properties(libstfhw PROPERTIES LINK_FLAGS "-L${CMAKE_BINARY_DIR}/../sim/liba3200nos -
L${CMAKE_BINARY_DIR}/../sim/libasfnos -L${CMAKE_BINARY_DIR}/../sim/nos ")
    target_link_libraries(libstfhw a3200nos asfnos nos-engine-link)
    message(STATUS "Set NOS Link Libraries")
ELSE ()
    message(STATUS "Set FLIGHT Link Libraries")
    /* set flight libraries here */
ENDIF ()
```

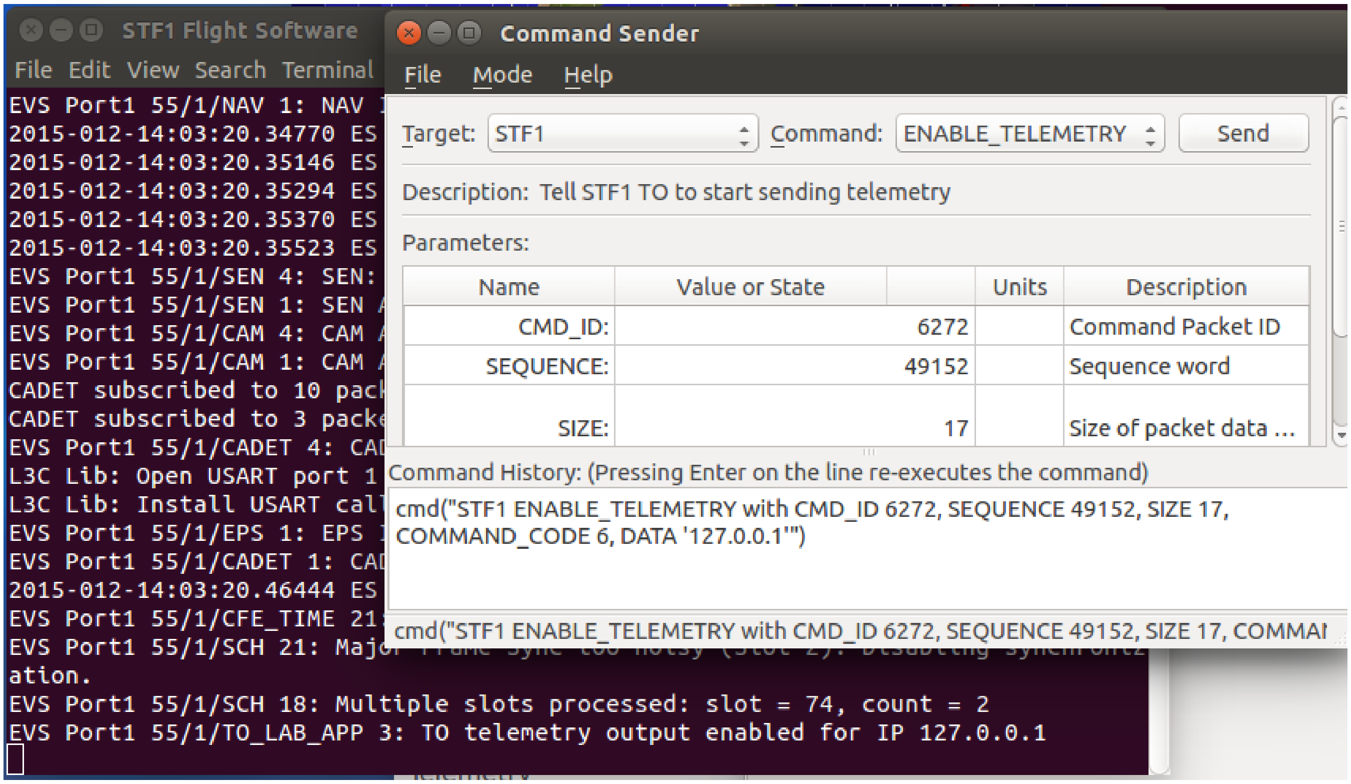
12 COSMOS Ground System

COSMOS is open source provided via Ball Aerospace[1] and is included with NOS3 to provide a ground station to the simulated spacecraft. COSMOS will also act as the official ground station for STF-1 at the Wallops Flight Facility providing the option to train operators. The telemetry and commands from cFS can be streamed in real-time, simulating current CubeSat capabilities such as the GloalStar network. One limitation is that data rate throttling is not currently supported.



This link to COSMOS is able to be completed thanks to two applications included in cFS. These are the command ingest (CI_Lab) and telemetry output (TO_Lab). The reason they are lab apps is due to the fact that they utilize UDP to communicate. The TO link is closed by default on start-up, but can be activated using a specific command packet. This is done by using the Command Sender tool in COSMOS. A special target was created named 'NOS3' with a single command to 'ENABLE_TELEMETRY'. Once sent, the TO_Lab 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_lab_sub_table.h' will be captured. Additional telemetry can be appended as necessary.

[1] Http://cosmosrb.com/



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_ID_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"
```

12.1 COSMOS File Creator

The COSMOS File Creator (CFC) is a tool that was originally developed at GSFC. The version included with NOS^3 contains updates from the ITC team at NASA IV&V. This tool has already been run and the output included with the release of NOS^3. It has been included in NOS^3 to allow for the development of other applications and regeneration of COSMOS command and telemetry files. The version of cFS included with the release also has all the necessary comments added to allow for proper generation. The tool is run directly from the command line with some additional arguments. Additional documentation and examples for use are provided in the NOS^3 release package.

13 Hardware In The Loop

Currently, four pieces of hardware are supported for communicating with NOS^3. These include the Aardvark, Bus Pirate, FTDI cable, and Raspberry Pi. Each of these will be explained in more detail stating all the protocols currently tested along with the installation and process to use. The term connector will be used to signify a program that bridges the gap between NOS Engine and the hardware allowing for this capability. These connectors do not function as real-time devices and as such a noticeable drop in through-put will be noted when testing. All files are located at 'nos3/sim/hwil/' and are broken down further into configurations for a specific protocol and the plugins (hardware) to be used.

13.1.1 Aardvark

The Aardvark can only be used if running NOS^3 on the host machine as pass-through to a virtual machine is not simply supported. A work-a-round to this has been found that utilizes NETCAT to forward this into the VM over the network then forward it again into a virtual COM port, but that will not be explained here. The Aardvark supports both I2C and SPI testing and is a useful tool for checking out hardware without integration into NOS^3. The integration allows for the verification of commands from the FSW to the device and telemetry returned.

13.1.2 Bus Pirate

The Bus Pirate was the first piece of hardware utilized to provide HWIL capabilities, and use has fallen off in place of more stable options such as the Aardvark and Wiring Pi libraries. The functionality is essentially the same as the Aardvark with both I2C and SPI support while purchased at a much lower cost due to the lack of a functional GUI and professional support. In future releases, this will be revisited and updated to match current standards.

13.1.3 FTDI Cable

An FTDI cable has been utilized to provide UART protocol support. This has only been utilized with ITAR simulators involving the Cadet UHF Radio. Due to this fact, it is not currently included in the NOS^3 release. More information is available upon request.

13.1.4 Raspberry Pi

The Wiring Pi library is to be used on a Raspberry Pi and has been tested on Rev 2B V1.1 solely. Instructions for the download and installation of this library are provided at the library website below:

http://wiringpi.com/

Currently, the build process limits the use of the Wiring Pi library to the CAM application supporting the ArduCAM Mini OV2640 with I2C and SPI protocols. The build flag for the wiring pi must be included in the 'nos3-build.sh' script. Simply append '-DWIRING_PI=YES' to the cmake command on line 10.

In order for the connector for find the appropriate library, the LD_LIBRARY_PATH must also be updated. Run the following commands in a terminal to achieve this:

"sudo su"
"export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/pi/Desktop/nos3-build/lib/"

Note that the appropriate communiations protocols must also be enabled in the 'raspi-config' prior to use.

13.1.5 Running the Connectors

In order to run a connector a new terminal must be opened and navigated to the 'nos3-build/bin' directory. From there the connector may be executed as the root user, 'sudo su', using the following example commands:

"./nos_i2c_connector –d 0"
"./nos_spi_connector –d 0"

Note that the device number may change depending on library type. See the help documentation for the specific connector for details.

Configuration files can be found at 'nos3/sim/hwil/__connector_type__/cfg/connector.cfg'. These files contain notes as what each field specifically means. Essentially, the bus, connector hardware, and address must be specified to allow for communication with NOS Engine.

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.

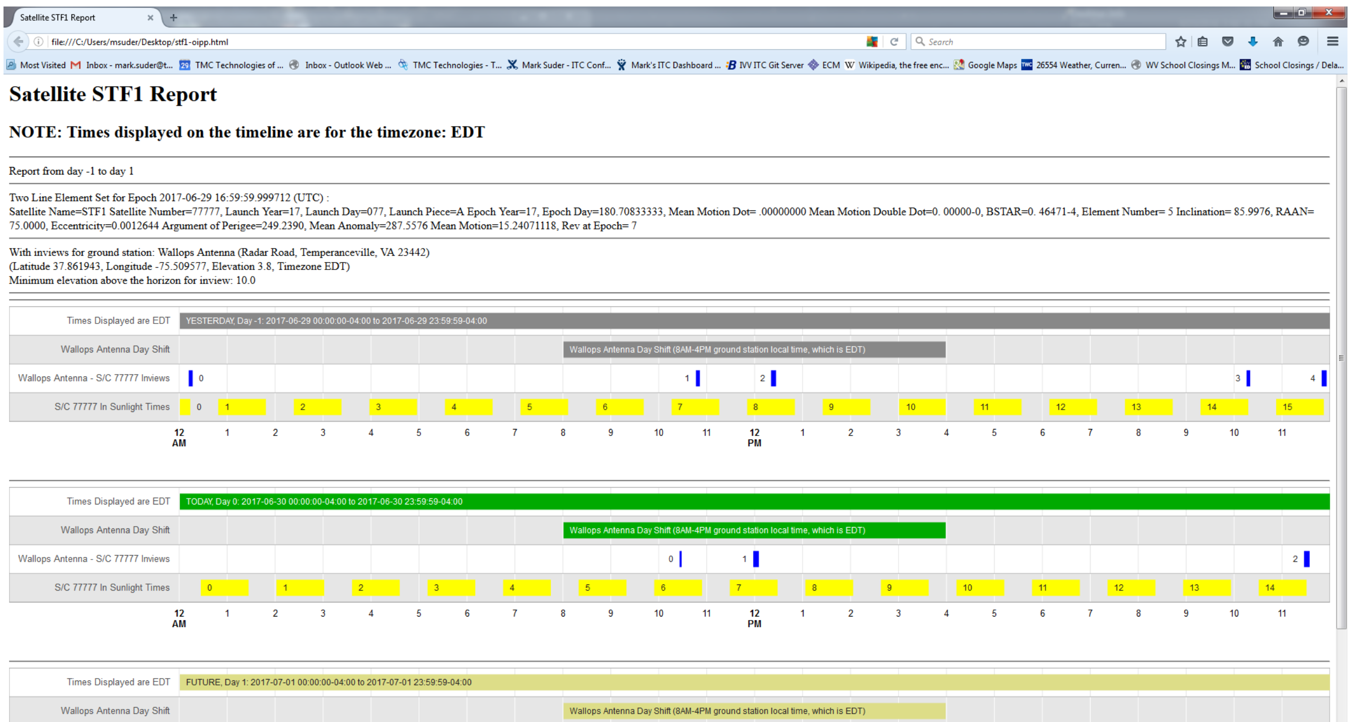


Figure 33 - Example OIPP Report

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*".