|  |
| --- |
| label_document |

OPS-SAT Software Simulator - Software User Manual

Table of contents:

1 Introduction 3

2 Acronyms 3

3 Architecture Overview 3

3.1 Integration to NanoSat MO Framework 3

3.2 Internal design 4

3.3 Commands 6

3.3.1 Commands execution 7

3.4 Templates 7

3.5 Command Results 7

4 Simulator Program Organization 8

5 Server 9

5.1 Logging 9

5.2 Simulator Configuration 9

5.3 Header 10

5.4 Scheduler 11

5.4.1 Validation rules 11

6 Client 11

6.1 Simulator data 13

6.2 Simulator header editor 13

6.3 Manual commands 14

6.3.1 Manual commands validation 15

6.4 Simulator devices 15

6.5 Simulator scheduler progress 16

6.6 Template editor 17

6.7 Simulator console 17

7 Orekit 18

7.1 Data files 18

8 Celestia 18

# Introduction

This document describes the operation of OPS-SAT software simulator standalone application.

# Acronyms

|  |  |
| --- | --- |
| CCSDS | Consultative Committee for Space Data Systems |
| MO | Mission Operations |
| SVF | Software Validation Facility |
| ICD | Interface Control Document |
| AGSA | Advanced Ground Software Application Laboratory |
| OREKIT | Orbital Extrapolation Kit (<www.orekit.org>) |
| ADCS | Attitude Determination and Control System |

# Architecture Overview

The scope of the simulator is running on platforms “Software Simulator” and “OPS-SAT”. The first one represents running the whole chain of provider/consumer on a single PC, the second represents running the MO framework apps on distributed machines, as intended in the decentralized design.

## Integration to NanoSat MO Framework

The simulator is part of the MO Framework services, as shown in Figure 1. It is instantiated at start-up and provides a data source to reproduce the real devices.

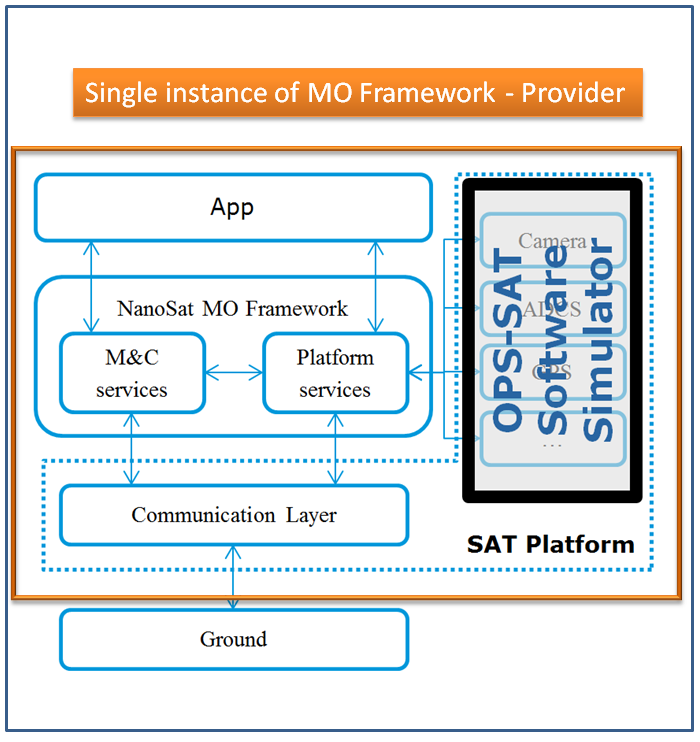


Figure 1 Platform Software Simulator Overview

## Internal design

The simulator aims to reproduce the complete command protocol of the following devices:

1. Fine Attitude Determine and Control System
2. GPS Receiver
3. High Definition Camera
4. Software Defined Radio
5. Optical Receiver

To achieve this, investigation is performed at level of ICDs for each of the devices. For example, the command protocol of Fine ADCS is I2C and the commands respect a common structure, as per Figure 3. The commands shown in Figure 4 are then mapped into a Java interface that’s designed to be functionally identical within byte level, see Figure 5. So there is a 1 to 1 mapping between device functionality and simulator functionality.

Figure 2 Organization of device classes

ICD

(BST FineADCS.pdf)

Simulator

Java Interface

IFineADCS.java

Generic Device

PFineADCS.java

implements

Additionally the simulator has provisioned commands for other boards on the OPS-SAT spacecraft, such as: CCSDS engine, Nanomind, MittyARM, FDIR, EPS.

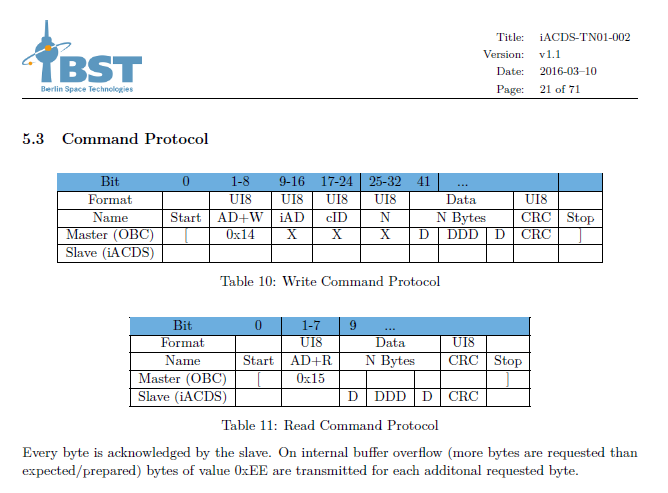


Figure 3 Command protocol structure FineADCS

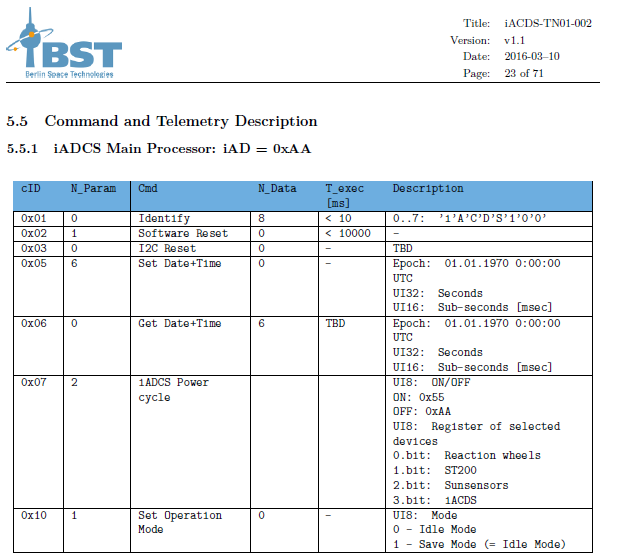


Figure 4 Command descriptions for given iAD and cID

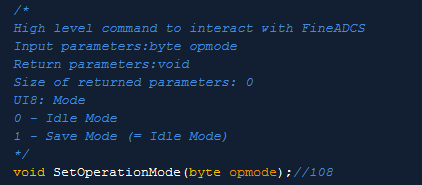


Figure 5 Interface representation in Java for a specific command method

## Commands

Each interface defined from the ICDs is a list of commands or the simulator operates with commands from the interface. All the commands have an internal ID, which is numbered according to the table.

|  |  |
| --- | --- |
| Device | Starting internal ID |
| FineADCS | 1001 |
| GPS | 2001 |
| Camera | 3001 |
| Nanomind | 4001 |
| FDIR | 5001 |
| SDR | 6001 |
| Optical Receiver | 7001 |
| CCSDS Engine | 8001 |
| MittyARM | 9001 |

At initialization, the simulator uses reflection to create a list of all the commands supported by the devices. This list will be sent to the GUI client program (see section 6) upon establishing connection. For example the GPS device contains a method called getNMEASentence:

@Override

@InternalData (internalID=2001,commandIDs={"",""},argNames={"inputSentence"})

public String getNMEASentence(String inputSentence) {

ArrayList<Object> argObject = new ArrayList<Object>();

argObject.add(inputSentence);

return (String) super.getSimulatorNode().runGenericMethod(2001,argObject);

};

### Commands execution

There are three ways to run commands on the simulator:

|  |  |
| --- | --- |
| Method | Comments |
| Direct calls to peripheral devices | The results (if defined) are returned to the caller function. If the command failed, the result will be null. String result=PGPS.getNMEASentence(“GPGGA”); |
| Manual call from GUI client | The results are returned to the GUI client, which displays them. If the command failed, the exception and/or error information will also be displayed. |
| Call from the simulator scheduler | The results are returned to the simulator scheduler itself, which discards them. |

For each of the three methods above, the simulator shall log to the file system all the command execution related information, as described in section 5.1.

## Templates

Each command can have a number of different input argument templates.

The input argument templates offer a convenient way to define specific requests to the methods. The listing below shows the definition of three templates that are connected to getNMEASentence (ID 2001).

2001|GLMLA|inputArgs=[String inputSentence={GLMLA}]

2001|GPALM|inputArgs=[String inputSentence={GPALM}]

2001|GPGGA|inputArgs=[String inputSentence={GPGGA}]

The scope of templates is using with the GUI client and part of the simulator scheduler.

## Command Results

A command result is the output of an executed command. It can be successful or failed. The simulator GUI client shows the complete information for each executed command in manual mode. The structure of a command result is as follows:

|  |  |
| --- | --- |
| Field | Description |
| Interface name | The interface inherited by the generic device |
| Method body | The actual method body of the command, i.e. byte[] runRawCommand(int cmdID,byte[] data,int iAD) |
| Internal ID | The internal ID of the method |
| Execution time | The real time on the simulator machine when the method was executed |
| Simulator time | The time of the simulator when the method was executed |
| Input arguments | The input data for the method |
| Output | The output data (if defined, for the method) |

For example calling the GPS getNMEASentence with position identifier GPGGALONG would return the following command result:

CommandResult{

* intfName=GPS,
* methodBody=String getNMEASentence(String inputSentence),
* internalID=2001,
* executionTime=Mon Jul 18 16:01:50 CEST 2016,
* simulatorTime=Thu Jul 13 16:23:41 CEST 2017,
* inputArgs=[String inputSentence={GPGGALONG}],
* output=[String ={$GPGGALONG,042341.216,4414.0420950,N,09019.0301373,W,1,0,0,650000.000,M,0,M,,\*XX}]}

# Simulator Program Organization

The application source tree can be found in:

*nanosat-mo-framework\mission\simulator\opssat-spacecraft-simulator*

The Maven project, package opssat.simulator.main consists of two main files:

1. MainServer.java
2. MainClient.java

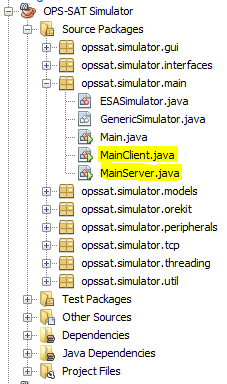


Figure 6 Maven project overview.

# Server

MainServer.java contains the starting point for the simulator class object, which is ESASimulator.java. This class inherits GenericSimulator.java.

It fulfils different roles:

1. Holds instances of classes for all the OPS-SAT simulated peripherals.
2. Creates a TCP listener socket for incoming connections from the GUI client. The default port on which the server listens is 11111. There can be up to 10 simulator classes running in parallel which will create sockets on ports [11111-11122].
3. Logs all the related information to the functioning of the simulator.
4. Keeps the simulator time.
5. Loads different settings from the file system.
6. Creates a TCP server for incoming Celestia applications connections (default port is 5909).

## Logging

The simulator creates a log file which records the information. It is located in $USER\_HOME/.opssat-simulator/. There are a few files created in this folder:

* Sim.log – overwritten each startup
* Cen.log – overwritten each startup
* Sim\_yyyyMMdd\_hhmmss – a new file is written on each startup following the date/time conventions
* Cen\_yyyyMMdd\_hhmmss – a new file is written on each startup following the date/time conventions

## Simulator Configuration

The simulator stores configuration information in files. There are three different files with following roles:

1. Header – contains general information like start/stop time, automatically start models and time, use optional modules (like Orekit,Celestia).
2. Templates – contains definitions for calling simulator commands (as defined in section 3.3).
3. Scheduler – contains a list of time-tagged command IDs with respective templates, to be executed within the simulator time.

At start up the simulator server will look it up in the current directory. For example, if the class MainServer.java is launched, the path will be the root of the directory.

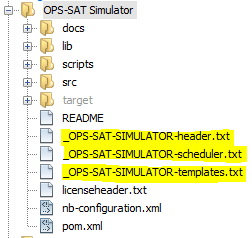


Figure 7 Maven project overview with configuration files highlighted

If the simulator is instantiated as part of a NanoSat MO Framework provider, the configuration files will be looked up in that folder, as seen in Figure 8.

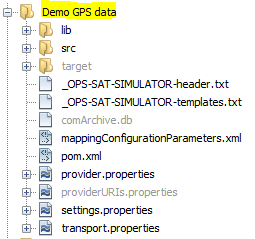


Figure 8 GPS Demo App running as provider, created configuration files

## Header

The following options can be modified from the header:

|  |  |
| --- | --- |
| Field | Comments |
| startModels=[true/false] | If true, the simulator will enable model loops for all devices (including time system) at startup. |
| startTime=[true/false] | If true, the simulator will enable time system keeping at startup. This setting is dependent on startModels. |
| orekit = [true/false] | If true, orekit class will be instantiated and used as a provider for simulation data (orbit, attitude, etc.). See section 7.1. Default is true. |
| celestia = [true/false] | If true, celestia server will be instantiated and used as a provider for satellite visualization. See section 8. Default is false. |
| updateFromInternet | If true, the simulator will try to connect to the Internet and retrieve updated information about GPS constellation TLEs. Default is false. |
| timeFactor = [1..1000] | The value of this parameter will be used as a multiplier for the passing of time. |
| startDate | Contains the start date/time for the simulation, in the format {2017:07:13 15:47:01 CEST}. |
| endDate | Contains the end date/time for the simulation. |

Note: Even if the simulator time passes the simulation end date, the program will continue to run.

## Scheduler

The simulator scheduler functions with the information parsed in the configuration file scheduler.txt. The typical structure of this file is as follows:

#days:hours:minutes:seconds:milliseconds|milliseconds|internalID|argument\_template\_name

A line which starts with # is ignored. A concrete line would then look like this:

00000:00:00:20:000|0000000000000020000|1001|CUSTOM

The information stored is time interval from start of simulator to running this command, kept in two formats, internal ID of the command and argument template. By using the internal ID and argument template effective run data is stored efficiently because the simulator will make a lookup to retrieve the actual input argument content for the specific template.

### Validation rules

The simulator will parse the entry for correct representation of time as well as existence of internal ID and argument template. This is the first check.

The second check is implemented because there are two possible ways to represent the simulator data, namely the:

1. days:hours:minutes:seconds:milliseconds
2. the milliseconds format

At the beginning a check will be made on both fields. If the time intervals are not equal, the first format representation which is different than zero (in this case priority is for above #1) will be accepted. The other format will then take the accepted value. If both values are zero, they shall be kept as they are and launched immediately at start.

Lastly, the third validation rule is that if the entries are not in chronological order, they will be reordered.

Whenever any of the above rules is not respected for at least one entry, the whole scheduler configuration file will be rewritten in a consistent way and a backup copy will be created in the same folder.

# Client

The GUI client offers the possibility to remote connect to a simulator instance and retrieve running information.

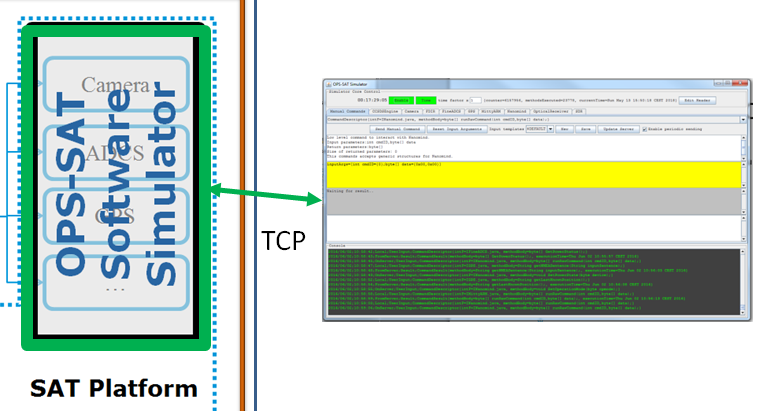


Figure 9 GUI client connection to simulator server.

It is a JAVA swing based GUI app which connects over TCP to an instantiated simulator server. At start, it is displaying the target IP address and port to which it is trying to connect:

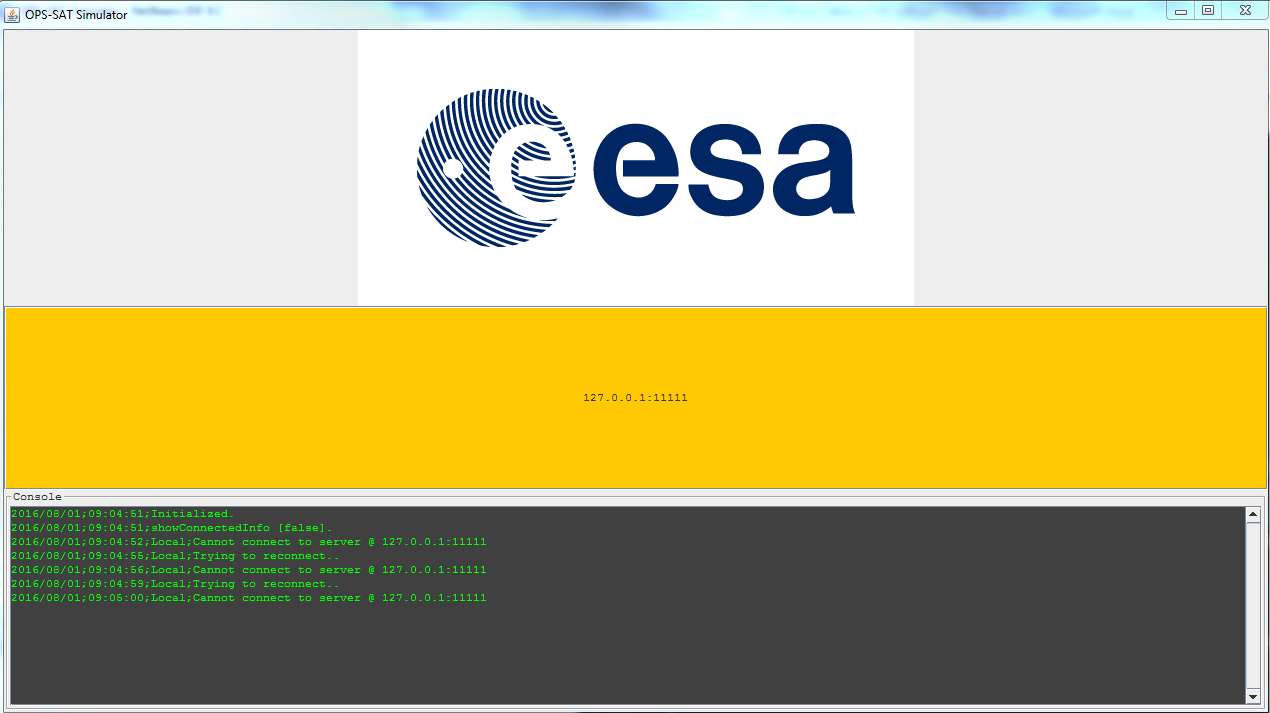


Figure 10 Default connection settings.

If the connection information should be changed (simulator running on a different machine with different IP and/or port), this can be done from the text input area.



## Simulator data

The simulator data is displayed at the top of the simulator window, in the frame called simulator core control.

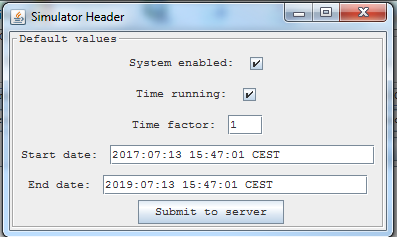


The information contained is related to the inner workings of the simulator. The user can see how much time has passed since the beginning, what is the active time factor, the number of computations done on the models (counter), the total number of methods executed (see section 3.3.1) as well as the current time. By clicking on the buttons enable and time it is possible to stop the execution of models and/or time.

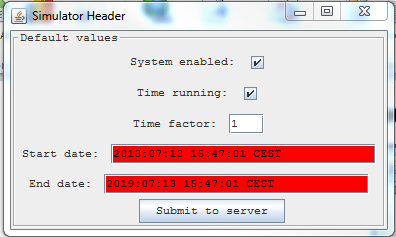


By clicking the Edit Header button, a separate window will appear.

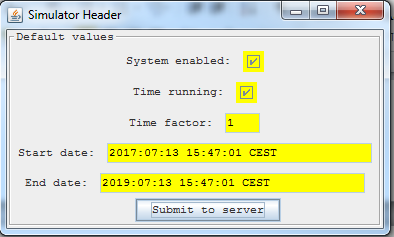
## Simulator header editor



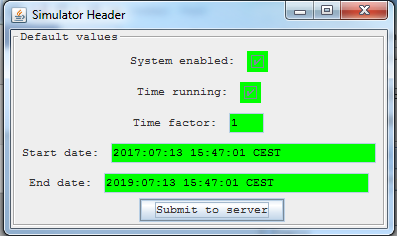
This window allows the user to change the default behaviour of the simulator with regard to the header (see section 5.3). If the data input is invalid, i.e. start/end dates are out of the allowed range or start is after end, the fields will be highlighted in red.



Once correct data is inserted (both start and end fields have white color), by pressing the “Submit to server” button, the user can save on the simulator’s header file the new settings. When the new settings have been forwarded to the server, the fields’color will be yellow:



Once the server receives and acknowledges the new simulator header, the fields will turn to green and the editor window will disappear after two seconds.



Note: With a successful submit to server action, the complete simulator internal state is reinitialized and reset.

## Manual commands

The manual commands tab offers the possibility to run specific commands from the entire available list.

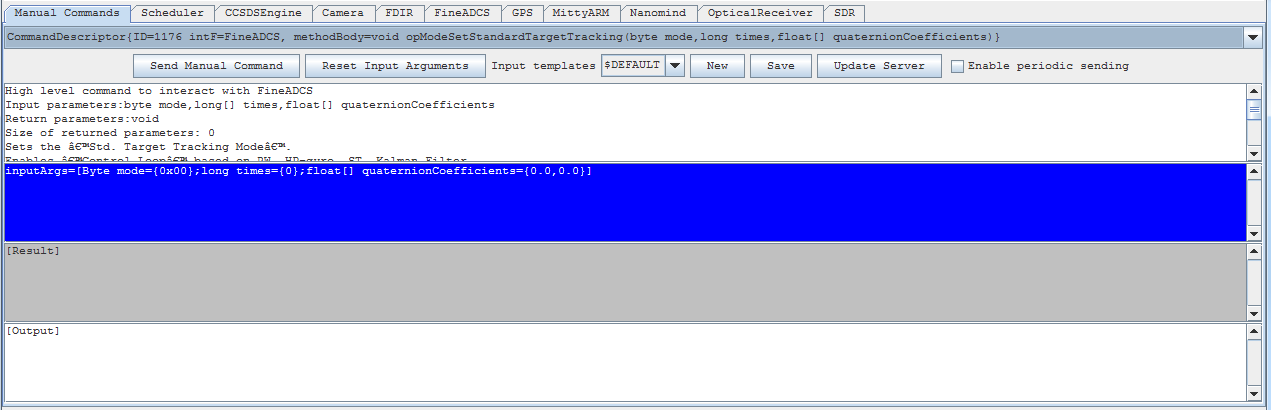


Figure 11 Manual commands tab overview

Below the tab selector is a combo box with all the available commands. Upon selecting one, the information displayed in the text areas below will be updated:

1. Description of method: includes information about input and output parameters
2. Input arguments: displays the input arguments in an easy readable form which allows changing values.
3. Result: will show the actual output of the executed command.
4. Output: will show a detailed information about the run of the command, as described in section 3.5.

### Manual commands validation

The input arguments are expected in a format which is generated in order to be user friendly: inputArgs=[int cmdID={0};byte[] data={0x00,0x00};int iAD={0}]. When the “Send manual command” button is pressed, the input string will be checked for following conditions:

1. Number of arguments matches expected one
2. Each argument is valid, depending on the data type

The two above checks are done locally and before actually sending the command to the simulator. If they are not met, the input arguments text area will become red and the output area will show the reason.



Figure 12 Missing input argument from expected command header.

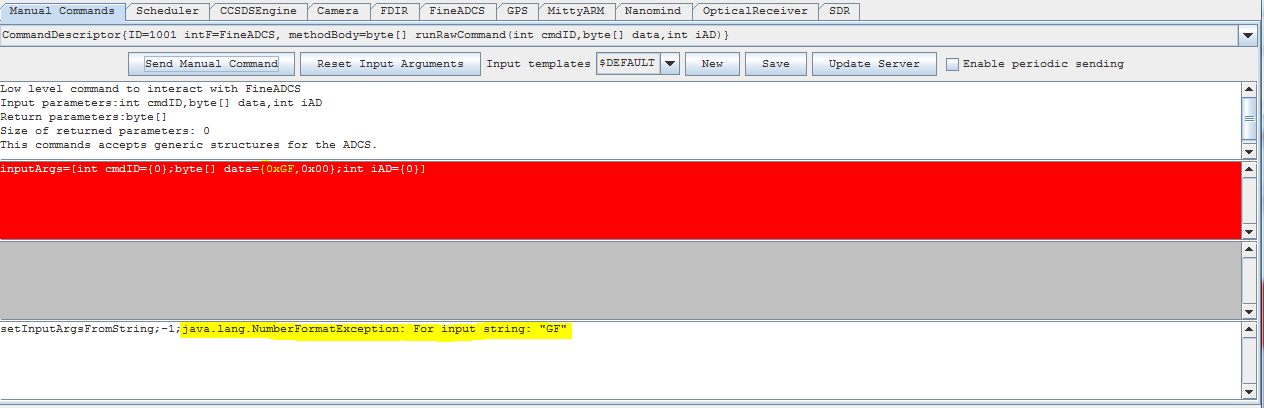


Figure 13 Invalid input value for byte parameter.

The “Reset Input Arguments” button will reset the inputArgs string to the default value.

## Simulator devices

The different tabs in the main window of the GUI client offer the possibility of switching between the data representations of simulated devices.



Inside the tabs there will be a selection of parameters which allow monitoring of the simulator model internal variables, like shown in Figure 14 and Figure 15.



Figure 14 GPS internal data representation.

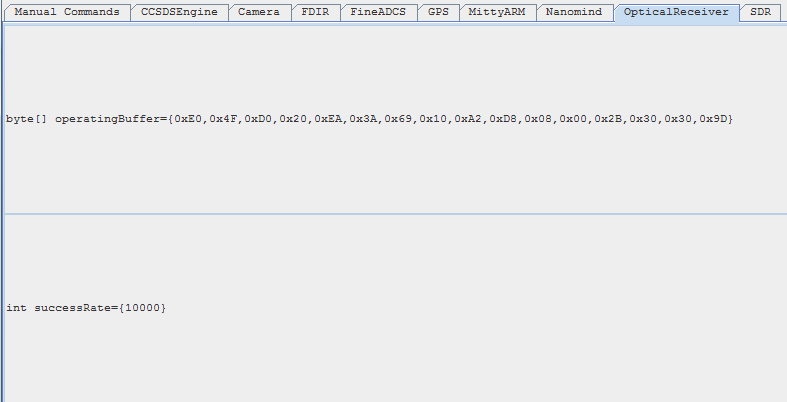
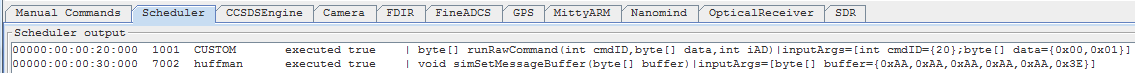


Figure 15 Optical receiver internal data representation.

## Simulator scheduler progress

In the scheduler tab it is shown the existing pending commands for the scheduler and their state of execution.



## Template editor

The manual commands tabs offers the possibility to edit and add new templates to each method.

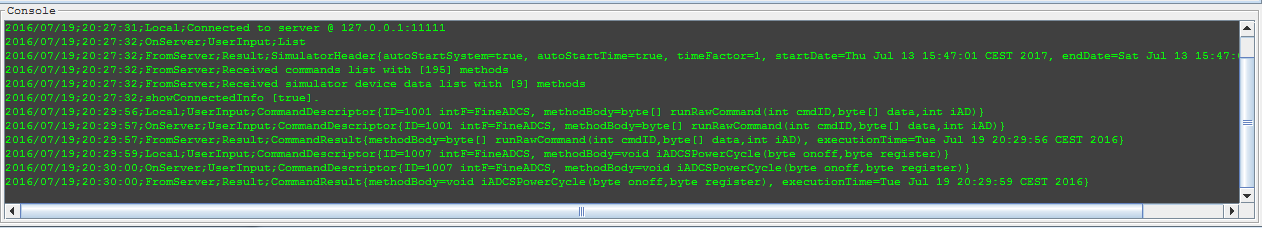


The initial template for each method is called $DEFAULT. This contains following values for the supported data types.

|  |  |  |
| --- | --- | --- |
| Data type | Default value / format | Value range |
| Byte | {0x00} | {0x00} to {0xFF} |
| Byte[] | {0x00,0x00} | {0x00} to {0xFF} |
| Int | {0} | {-2147483648} to {2147483647} |
| Int[] | {0,0} | {-2147483648} to {2147483647} |
| Long | {0} | {-9223372036854775808} to {9223372036854775807} |
| Long[] | {0,0} | {-9223372036854775808} to {9223372036854775807} |
| Float | {0.0} | {1.4E-45} to {3.4028235E38} |
| Float[] | {0.0,0.0} | {1.4E-45} to {3.4028235E38} |
| Double | {0.0} | {4.9E-324} to {1.7976931348623157E308} |
| Double[] | {0.0,0.0} | {4.9E-324} to {1.7976931348623157E308} |
| String | {} | {} |

## Simulator console

The simulator console provides a list of messages which aid the user into understanding the program function and interaction between server and client.



The types of messages displayed are as follows:

|  |  |  |
| --- | --- | --- |
| Source | Type | Comment |
| Server | OnServer | A specific event has occurred on the server. Frequently this is an echo to a command that has been initiated by the client or the NanoSat MO framework parent class. |
| Local | Local: Connected | The client has successfully established connection to the simulator server. |
| Server | FromServer: Received | Data has been received from the server, this can be simulation header, commands list or commands result |
| Local | UserInput | The user has taken action from the control interface, such as starting/stopping time, changing the time factor, changing the simulator header, adding a new simulator template |

# Orekit

This module provides orbital and attitude information in a variety of coordinate frames.

## Data files

The Orekit library implementation requires a variety of files:

1. Ephemeris information for different common space bodies.
2. Magnetic model coefficients
3. GPS constellation TLE files

At launch, the orekit core constructor will check if each data file exists and if not, will extract it and copy to the target folder shown in the Table 1.

|  |  |
| --- | --- |
| OS | Path |
| Windows | C:\Users\<Username>\.ops-sat-simulator/data |
| Linux | ~/.ops-sat-simulator/data/ |

Table 1 Location of resources folder

Should the file not be found, the simulator will bypass the use Orekit setting and switch to the internal provider, while providing this error output message:

orekit initialization failed from [org.orekit.errors.OrekitException: no IERS UTC-TAI history data loaded]! Switching module off

# Celestia

Celestia application provides interactive visualization of satellite position and attitude.

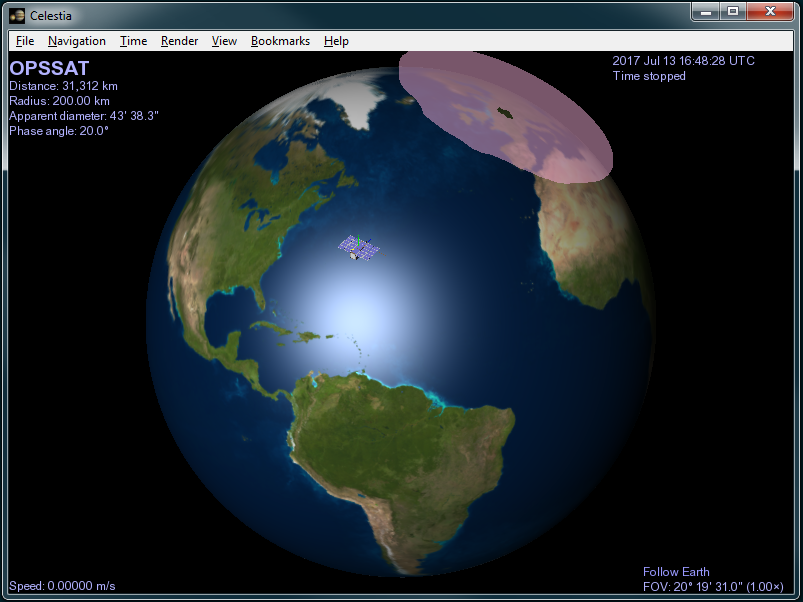


Figure 16 Celestia visualization tool.