Servo Controller Simulation Manual

Revision 1.0

May 6, 2010

Bernard Jzexoia Avery
Samuel Mallon
Jared Pitterle
Talbert Tso

Contents

1 Introduction **2**
1.1 High-Level Overview . . . 2

2 Servo Controller Simulation **5**
2.1 Installation . . . 5
2.2 Interface . . . 5
2.2.1 Actual Results Panel . . . 5
2.2.2 User Input Panel . . . 8
2.2.3 Expected Results Panel . . . 10
2.3 Implementation . . . 11
2.3.1 Physics Model . . . 11
2.3.2 Serial Communications . . . 13
2.3.3 Graphical User Interface . . . 14

3 MEU Simulator **16**
3.1 Installation . . . 16
3.2 Interface . . . 17
3.3 Implementation . . . 17

A Acronym Glossary **18**

B Serial Communications Packages **18**

1 Introduction

This document details the operation and implementation of the Servo Controller Unit (SCU) simulator in **Section 2** and the associated Modular Electronics Unit (MEU) simulator in **Section 3**. Building and installation instructions are also provided in **Sections 2.1** and **3.1**. Any acronym used in this document can be found in **Appendix A**.

In addition to this document, there is a complete set of javadocs for the code in the project, and all code in the project is commented. There is also a website detailing the project at http://www.cefns.nau.edu/Research/D4P/EGR486/CS/10-Projects/LMPF/index.html.

The members of Team HAWC can be reached via the following e-mail addresses:
• Bernard Jzexoia Avery bc229@nau.edu
• Samuel Mallon srm224@nau.edu
• Jared Pitterle Jared.Pitterle@gmail.com
• Talbert Tso Talbert.Tso@gmail.com

1.1 High-Level Overview

To better understand how the simulators are organized and structured, the following overview describes the architecture of the program and how data flows within the program.

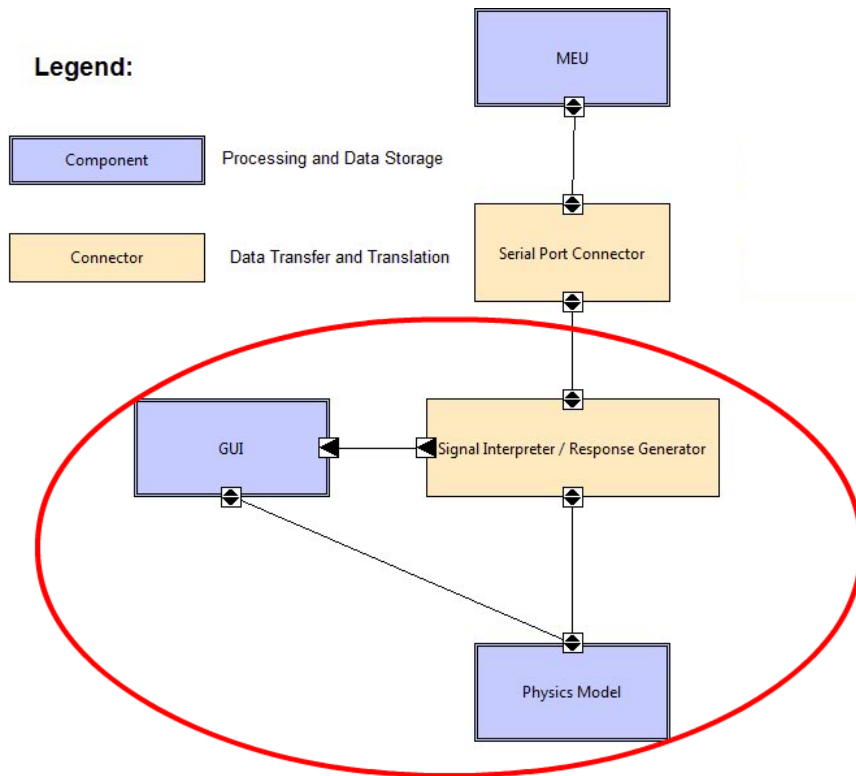
There are three primary components to this application:

1) The Graphical User Interface (GUI)

2) The Signal Interpreter and Response Generator

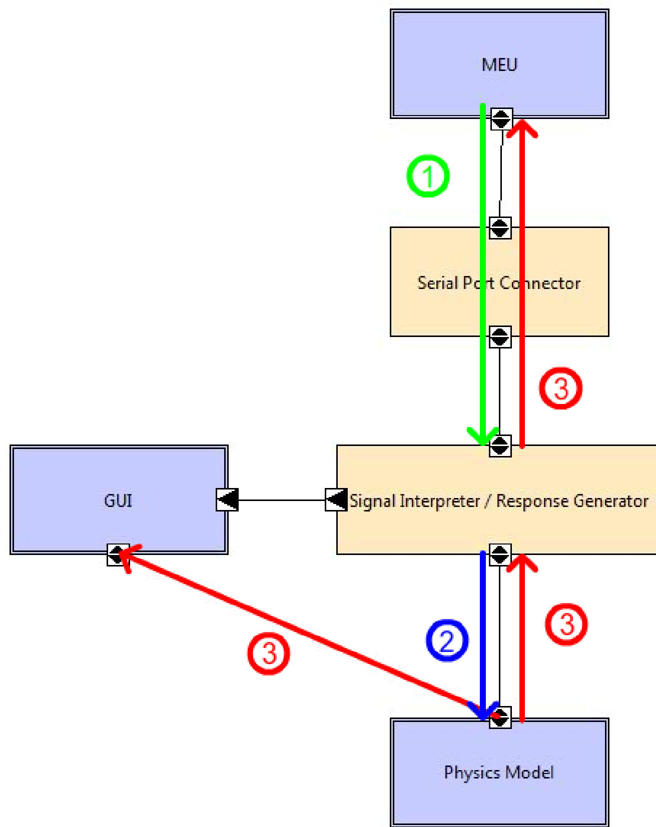
3) The Physics Model

Figure 1: High-level program architecture



Each of these components plays a key role in the operation of the system. The physics model calculates and stores real-time data about the state of the gimbal assembly, and contains methods and commands for each of the possible motions and actions of the device itself. The Signal Interpreter and Response Generator is responsible for translating command received over the serial port into usable methods for the Physics Model, and also for crafting correctly constructing response messages based on the status objects returned by the Physics Model. The third component, the GUI, is responsible for conveying information about the real-time state of the simulation as well as the contents of the last response message sent to the MEU to the engineer using the program. The GUI also contains input features allowing the engineer using the program to select the appropriate serial port to listen on or set custom error flags for the program to simulate.

Figure 2: Data flow through application



The figure above shows an example of the flow of data through the program, in this case in response to a command from the MEU.

1) A command is sent by the MEU over a serial connection and into a serial port on which the signal interpreter is listening.

2) The Signal Interpreter interprets the command into a java method call and calls the appropriate method from within the Physics Model. This will cause an action, such as changing a soft stop or causing the gimbal to move.

3) The Response Generator will then formulate and send an automatic response message by getting the state of the Physics Model and composing it into an appropriate byte-format response message which can be interpreted by the MEU. At the same time, the same status message is sent from the Physics Model to the GUI (in the form of a java object rather than as a byte stream) and displayed. This allows the engineer using the program to quickly view the contents of the last status message sent by the SCU simulator.

4) Another process which goes on in parallel with the flow of data just described is that the GUI will continuously pull the status of the Physics Model and display that status 120 times per second. This allows the engineer using the simulation to view the immediate status and status changes of the physics model in real time.

2 Servo Controller Simulation

The SCU Simulator is an application which will simulate the behaviors of the 2-axis SCU to be used in conjunction with the MEU on an ARL-M aircraft. This simulation can be installed and run on nearly any computer with a serial port, and has been tested on both Windows and *nix operating systems.

The purpose of this simulation is to serve as a complete replacement for the physical SCU and associated gimbal-antenna assembly for the purposes of testing and development of the MEU or other devices which may need to interact with the SCU at a future date. When properly installed and connected to the MEU or a similar device via a serial port connection, this simulation will receive commands just as the physical device would, and respond accordingly by simulating changes in state, motion, and position and by sending the appropriate response messages back over the serial connection, just as the physical device would.

2.1 Installation

Before installing the servo controller simulator, the following programs should be installed:

- JDK (v1.6 or higher)
- Apache Ant (any version)
- `javax.comm` package (see **Appendix B**)

To install the servo controller simulator:

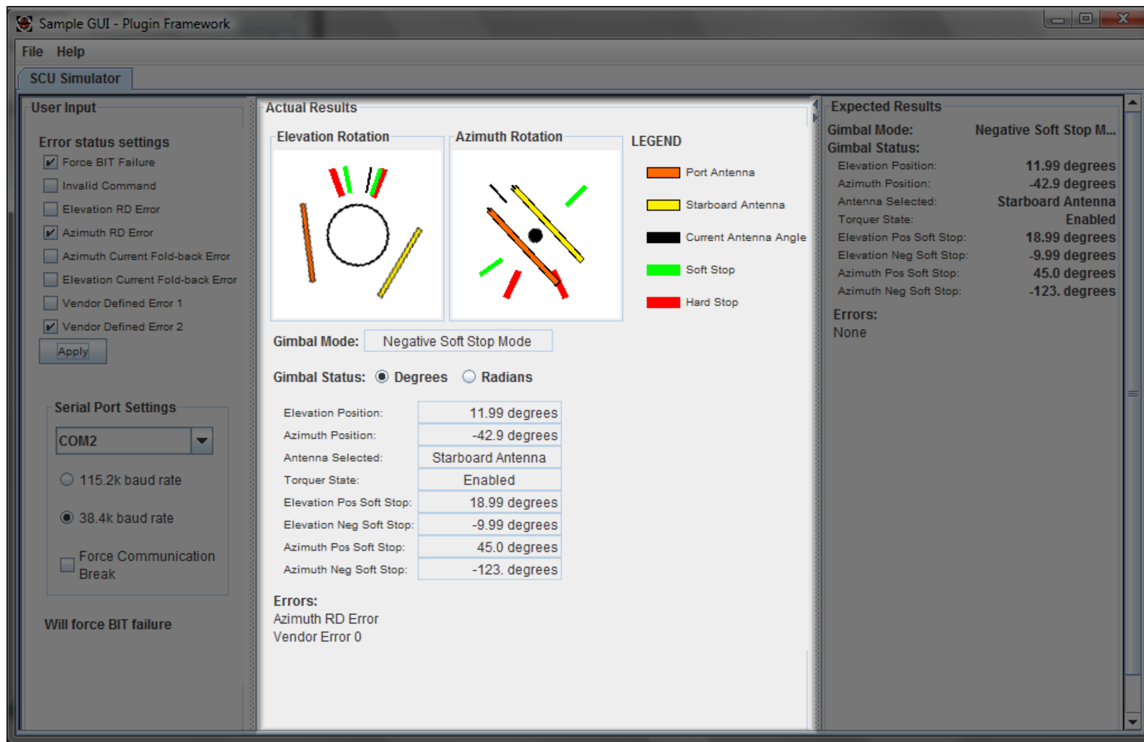
1) Open a terminal at the directory containing the build.xml file and run the command "`ant scu`" without the quotes
2) This will create a file named ServoControllerPlugin.zip which contains a plugins directory with the servo controller plugin contained in its own directory, ServoController, inside the plugins directory.
3) Copy the ServoController directory to the framework plugins directory
4) Run the plugin framework and use the servo controller simulation

2.2 Interface

2.2.1 Actual Results Panel

The Actual Results Panel, shown in **Figure 3**, shows the current state of the physical simulation at any given point in time, through a combination of graphical, numerical, and textual outputs. It updates 120 times a second to give constant feedback on the current state of the physical simulation.

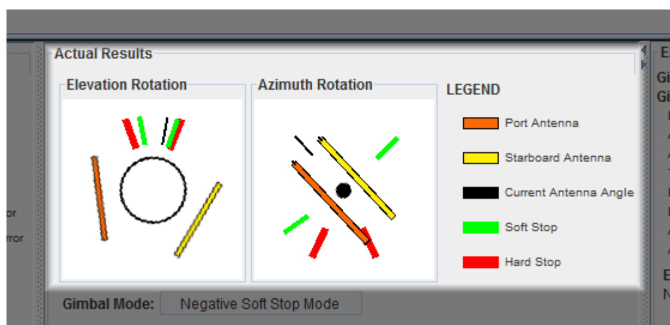
Figure 3: The Actual Results Panel



Displayed Controls and Information

The Actual Results Panel has a number of fields displaying various aspects of the current state of the simulation. The top-most section is a graphical display of the current positioning of the antenna simulation, as seen in **Figure** 4. These visual displays show the current angle in elevation and azimuth, as well as the hard and soft stops in both directions. The left panel here is a forward-facing view that depicts the elevation rotation of the simulated antenna. The graphic on the right is a top-down view that depicts the azimuth rotation. As shown in the legend on the right, an orange bar surrounded by a black border represents the port antenna, a yellow bar surrounded by a black border represents the starboard antenna, the small black line indicates the current angle of the antenna assembly, and the green and red lines represent the current positions of the soft and hard stops, respectively.

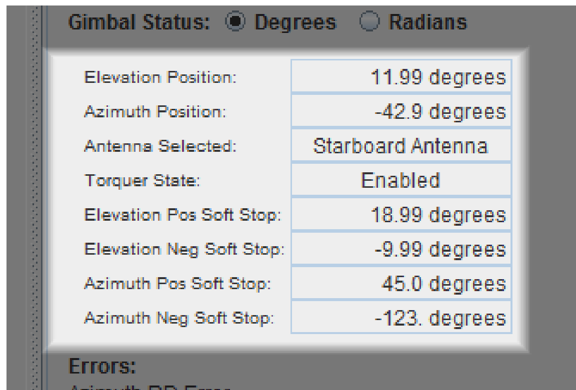
Figure 4: The graphical display of the antennas' state in the Actual Results Panel



Below the graphical representation is a combination of textual and numerical output that represents the current state of the simulated antenna, shown in **Figure** 5. The current "mode" that the gimbal is in is

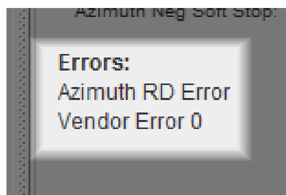
reported here, along with the antenna position in elevation and azimuth, positive and negative soft stops in both directions, the current state of the torquers (either enabled or disabled), and a reading of which antenna is currently selected (either port or starboard). A user can choose either the “Degrees” or “Radians” radio button, which will cause units to be displayed in either degrees or radians, respectively.

Figure 5: The numerical data of the antennas’ state in the Actual Results Panel



The final display on the Actual Results Panel is the “Errors” section, which is simply a list of the errors being reported in the simulation **Figure** 6. These errors can be toggled using the checkboxes in the User Input Panel.

Figure 6: The error status of the servo controller in the Actual Results Panel



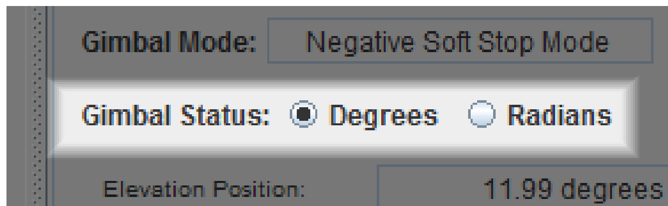
Usage

The main purpose of this panel is to show the current state of the simulation, and is composed primarily of output components. Outputs can be read visually from the graphical sub-panels at the top of this panel, or from the various numerical and textual displays below. This output changes whenever the state of the simulation changes as a result of commands sent in the proper format over the serial connection, either from the MEU or some other device or program capable of sending the same commands.

The errors currently being reported by the simulation are also listed in this panel. The reporting of these errors can be toggled using the User Input Panel.

The one form of input available are radio buttons to change the units the numerical outputs are displayed in. Clicking either the “Degrees” or the “Radians” radio button, as seen in **Figure** 7, will change both the Actual Results Panel and the Expected Results Panel to display values in the corresponding units.

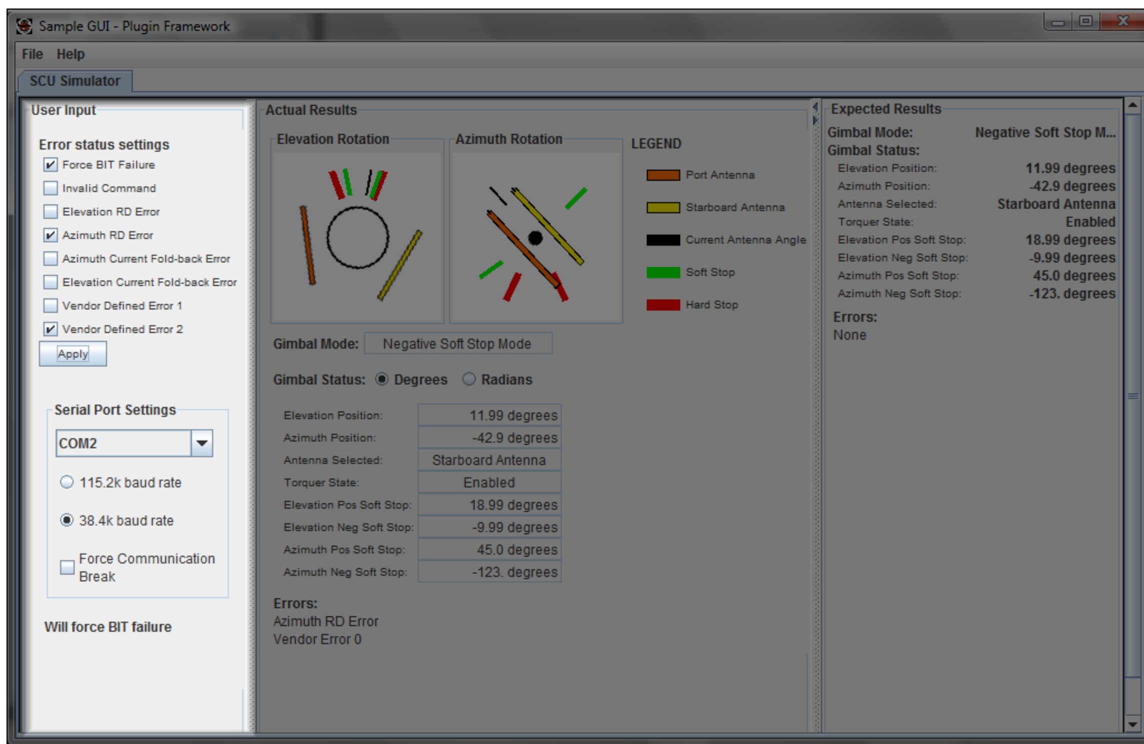
Figure 7: The radio buttons to change the units of the numerical information



2.2.2 User Input Panel

The User Input Panel allows the toggling of errors to be reported in the physical simulation, as well as selection of serial communication settings. The primary purpose of this panel is to specify settings for the simulation that the MEU is not capable of setting through standard commands.

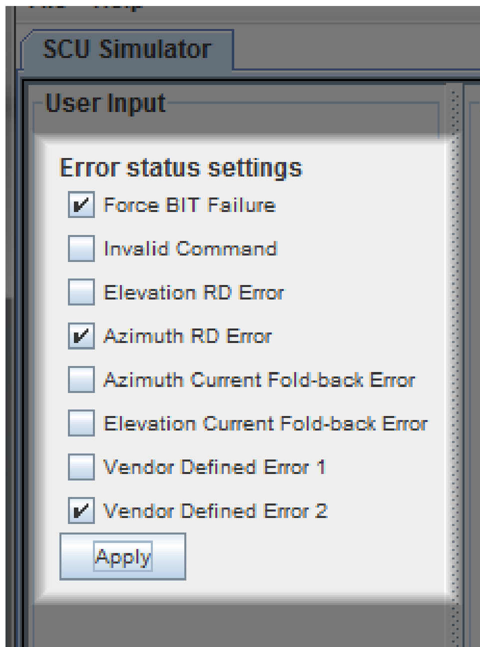
Figure 8: The User Input Panel



Displayed Controls and Information

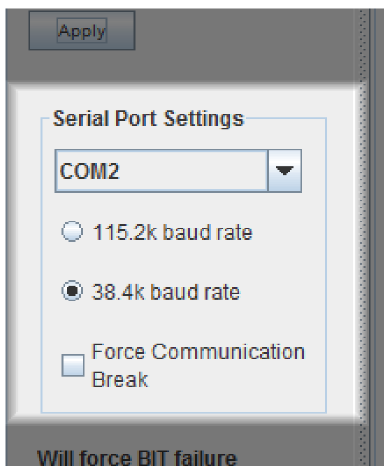
The User Input Panel shows which error messages are currently set to be displayed in the simulation (**Figure** 9). The selected errors are not reported in the simulation until the Apply button is pressed, so the Actual Results Panel should be observed to determine which of these errors have actually been applied.

Figure 9: The error selection checkboxes in the User Input Panel



This panel also shows which serial communication port is currently selected, as well as the baud rate for this communication, as seen in **Figure** 10.

Figure 10: The serial settings portion of the User Input Panel



Usage

Any combination of errors can be selected from the "Error status settings" section at any time, and applied in the physical simulation by pressing the "Apply" button (**Figure** 9 above). Clicking the checkboxes for the corresponding error settings, or pressing their shortcut keys (Alt+1 through Alt+8 on Windows, or Option+1 through Option+8 on Mac), will toggle the reporting of that error. The "Apply" button must be pressed for any changes to take effect.

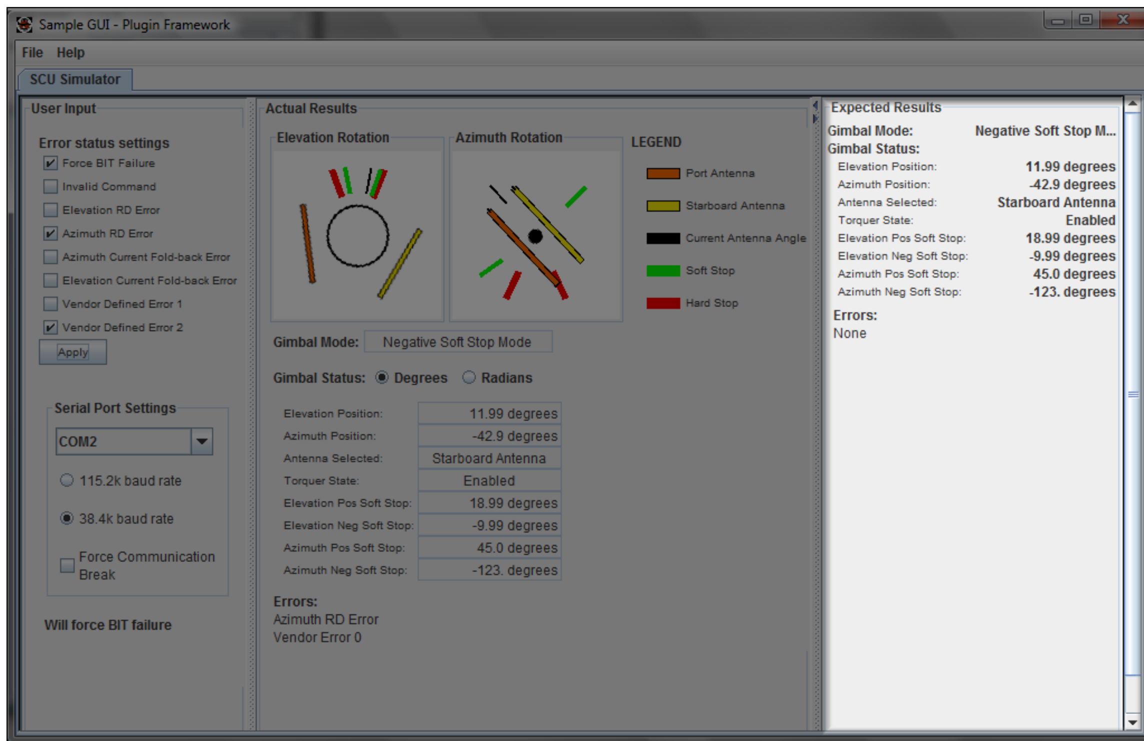
The "Serial Port Settings" section allows for selection of a communication port, using the drop-down box, and a baud rate setting, using the corresponding radio buttons (**Figure** 10 above). These settings are

applied upon selection. The “Force Communication Break” checkbox is used to cause an artificial break in communication. When this box is checked, the simulation will ignore all incoming commands, and will not send status responses.

2.2.3 Expected Results Panel

The purpose of the Expected Results Panel is to show the current status being reported to the MEU. The panel updates every time a status message is sent from the simulation as a result of it receiving a command from the MEU. This panel only takes information from status messages, and never directly from the simulation itself. Because of this, it will only update when a new status message is sent to the MEU.

Figure 11: The Expected Results Panel



The Expected Results Panel displays the state of the simulation according to the most recent status message sent to the MEU (**Figure** 11). This panel updates every time a status message is sent, and displays the current state through a combination of numerical and textual outputs. The values representing the state include the current “mode” the gimbal is in, the elevation and azimuth positions, the currently selected antenna, the state of the torquers, and the negative and positive soft stops in both elevation and azimuth directions. The units in which the values are displayed can be set using the “Degrees” and “Radians” radio buttons on the Actual Results Panel (**Figure** 7).

In the “Errors” section at the bottom (**Figure** 11), the Expected Results Panel also displays the errors reported in the most recent status message.

Usage

The only function of the Expected Results Panel is to display the contents of status messages, and these outputs can be read from the corresponding fields. Although there are no input options in this panel, a user

can toggle it between being hidden and visible by clicking the small arrows on the upper left border of the panel (**Figure 12**). A user can also choose whether units are displayed in degrees or radians by choosing from the "Degrees" or the "Radians" radio buttons, respectively, on the Actual Results Panel (**Figure 7**).

Figure 12: The hide panel button for the Expected Results Panel

2.3 Implementation

2.3.1 Physics Model

The physics model is responsible for calculating and recording all information on the current state of the servo controller, including position information and all errors.

Central Interface

The hub of all activity within the physics model, the Central Interface is represented by the class `PMCentralInterface` and contains all the necessary data and components to simulate the motions and behaviors of the gimbal-antenna assembly. `PMCentralInterface` is a singleton class and `getInstance()` should be called to retrieve the singleton instance.

The primary role of the central interface is to interpret the commands which the SCU can receive (such as "Position Mode," "Cage Mode," "Disable Torquers," etc.) into individual actions which are required of lower-level components such as servos, sensors or antennas. The central interface is also responsible for storing the mode the device is currently in, errors the device is currently simulating, and for simulating the power-up sequence. The central interface is also responsible for gathering all necessary data from lower-level components to compose the status response messages which the simulation will send out in response to commands from the MEU.

The Central Interface contains instances of several lower-level components, most notably two instances of `Servo`, which do the bulk of all analysis and simulation work in the program. However, these components are specifically intended to work within the Central Interface, so in general it is best to simply instantiate and call methods on `PMCentralInterface` directly rather than on the lower-level components individually.

The Central Interface contains methods which represent each of the commands which the SCU can receive and methods to initialize the various error states which the simulation can simulate. These methods are all fairly straightforward in their execution and are described in more detail in the javadocs for this program. However, one of these methods, `getStatus()`, is somewhat more complex and deserves a deeper explanation.

`getStatus()`

`getStatus()` is a method which will, logically enough, return a `Status` object representing the state of the device. However, this method will also update the status in the process of gathering it. There is no other public method which will cause the status of the device to update[1], and so `getStatus()` should be called at least 50 times per second in order to maintain an accurate simulation. In this application, the GUI calls this method 120 times per second and uses the results of the message to display the real-time status of the device.

The `getStatus()` method works by gathering the various properties of the `Servo`, `Antenna`, and `TempSensor` objects contained in `PMCentralInterface` into a single `Status` object and then returning that object. In the process of doing so, it will call the method `Servo.getPosition()` which will be described later in this

[1] Updating the status is different from causing the state to change, which happens any time a command is received

document. This ensures that the Servos will be in an updated position before the status is returned, and as long as the application running the simulations calls `PMCentralInterface.getStatus()` at least 50 times per second, there is no need to call `Servo.getPosition()` (which updates the status of the servos as well as returning the position) directly.

In the process of updating the simulation state, `getStatus()` also handles two other aspects of state change:

- The first is the power-up sequence. Once a power-up sequence is started, the starting time of that sequence is recorded. When `getStatus()` is called, it checks to see if it has been at least 5 seconds since entering the power-up state, and if so will end the power-up sequence. This sequence will succeed unless the `forcePowerUpFail` flag has been set.
- The second is Cage Mode. When the SCU receives a Cage Mode command, it will **not** report its mode as "Cage Mode" until it actually reaches the home position. The getStatus() method, when called, will check to see if the device has recently been issued a "Cage Mode" command by checking the `cageMode` flag. When the `cageMode()` command is received, this flag is set to true but the mode of the device is not changed. If this flag is true, then the device will enter "Cage Mode" once the home position is reached. The `cageMode` flag will be set to `false` once the home position is reached or if any command other than "Cage Mode" is received.

Servos

The purpose of the `Servo` class is to simulate the motions of the servos for each axis of the device. Servos maintain information about their current position, mode, rate of motion, and soft and hard stops. The Servo class can calculate its position based on the current mode of motion and the state of the Servo when the last command was issued.

While a complete listing of methods can be found in the javadocs of this program, those methods which involve the motion of the device deserve a deeper description, which will be provided here.

`upperBound()` and `lowerBound()`

The `upperBound()` and `lowerBound` methods are not terribly complex but are required in order to understand the methods which are about to be described. These methods will simply return either the soft stop or hard stop in the posititve and negative directions respectively, depending on which of these presents the more immediate barrier to the motion of the device.

`setPosition(double)`

A servo has only 2 modes of motion, "Position Mode" and "Rate Mode" ("Cage Mode" is a special case of "Position Mode"). `setPosition(double)` will instruct the servo to move to a specified angle at its default rate, then stop. When this method is called, the servo will record that it is in position mode by setting the `rateMode` flag to false, recording its current position in a special property called `startAngle`, and will record the current time into `startTime`. These values allow the `getPosition()` method to calculate the current position of the servo. "Cage Mode" is accomplished by sending both servos a `setPosition(0.0)` command. The only complexities of this method come into play when the hard and soft stops are involved.

If we assume that the servo is currently in a valid position between the upper and lower bounds of motion, then if a position command is issued with a desired angle which is beyond either of these bounds, the servo will reinterpret this command as a command to move to the nearest valid position. In other words, if this method is called using an angle which is above the upper bound of the device, then the servo will reinterpret this as a `setPosition(double)` command with the upper bound as the desired angle.

The servo will never move into an invalid position as the result of any of its motion commands. However, because soft stops can be changed arbitrarily and at any time, it is possible for the servo to be in an invalid position due to the soft stops being set to positions that would cause the current position to be out-of-bounds. In this event, the servo will still respond normally to any `setPosition(double)` command which would move it towards the valid range of motion, whether or not the angle requested is actually within that range.

If the soft stops have been set such that the negative soft stop is higher than the positive soft stop, then there is no valid range of motion and `setPosition(double)` will have no effect. `setPosition(double)` will also have no effect in the event that the servo has been disabled.

`setRate(double)`
The `setRate(double)` method will instruct the servo to move at a specified rate between 30 and -30 degrees per second. The device will move at the specified rate until the upper or lower bound has been reached or until another command is issued. In the event that the servo is outside the valid range of motion when the command is received, the servo will respond to this command only if the specified rate would move the servo towards the valid position, not if it would move the device farther away from that range.

Just like `getPosition(double)`, this method will record the current position, the time the command was issued, and the specified rate which will enable `getPosition(double)` to accurately calculate the position of the device at any time. If the servo is disabled or if there is no valid range of motion due to the position of the soft stops being inverted, then this device will not cause any motion to occur.

`getPosition()`
This method will both update and return the current position of the device. It will do this by looking at the `rateMode` flag to determine if the servo is moving in rate mode or position mode. Based on this, the servo will calculate its position by multiplying the rate of motion (equal to the default rate if in position mode) by the time passed since the command was sent and adding this to its initial position at the time of the last movement command. Further controls will stop the servo if it reaches an upper or lower bound. Since this method updates the position of the device, it should be called at least 50 times per second for an accurate simulation. This method is called by `PMCentralInterface`'s `getStatus()` method, so there is no need to call this method directly if `getStatus()` is already being regularly called elsewhere. The GUI of this simulation will call `getStatus()` 120 times every second.

Temperature Sensors

Represented by class the class `TempSensor`, temperature sensors can be initialized with any temperature value, and will always return this value. These do not simulate any sort of temperature reading and the only complexity to the class is that the `getTemperature()` method ensures that only values within the specified range of temperatures a status response can store will be returned.

Antennas

The only information stored by the antennas is whether or not they are selected. In this simulation, only one antenna can be selected at a time. However, this is handled by `PMCentralInterface` and is not a feature of the antenna class.

2.3.2 Serial Communications

Serial communications are implemented in the `SignalInterpreter`, `SerialConfiguration`, and `ResponseGenerator` classes.

Signal interpreter

All information that is sent or received on a serial port occurs in the `SignalInterpreter` class. The primary methods in `SignalInterpreter` are `enableCommunication()`, and `write(byte[])`. The method `enableComunication()` starts an asynchronous thread which reads information from the serial port and processes the information into commands that can be issued to the physics model. The `write(byte[])` method is used to send information over the serial port.

Other important methods in `SignalInterpreter` are `enumeratePorts()`, `setSerialPort(CommPortIdentifier)`, `setSerialConfig(SerialConfiguration)`, `disableCommunication()` and `forceCommunicationBreak(boolean)`. The method `enumeratePorts()` initializes the serial drivers and lists all available serial ports on the system. The method `setSerialPort(CommPortIdentifier)` sets the serial port to use for communicating, but does not start serial communications on that port. The method `enableCommunication()` must be called to start communicating on the selected port. The method `disableCommunication()` is used to stop communicating on the currently selected serial port and is called any time `setSerialPort(CommPortIdentifier)` is called. The method `forceCommunicationBreak(boolean)` is used to simulate an actual break in serial communications. The serial communications are still operational, but all information on the serial port is ignored.

Response Generator

The `ResponseGeneartor` class is used to retrieve status information from the physics model and send the status information over the serial port. `ResponseGeneartor` uses `SingalInterpreter`'s `write(byte[])` method to accomplish this. `ResponseGeneartor` also sends status information to the GUI any time status information is sent over the serial port. The method used to accomplish this is `sendCurrentStatus()`.

Configuration

The `SerialConfiguration` class is a Java enum type that describes the baud rate, number of stop bits, number of data bits and number of parity bits for communicating over a serial port. This class is used to set the current serial configuration for communications in the `SignalInterpreter` class. The enum constants are named `_baudrate_data_parity_stop` where `baudrate` is in bytes per second, and the others are the number of bits used for each, except in the case of no parity bits, where "`N`" is used instead. Only the two possible configurations for the servo controller are given.

2.3.3 Graphical User Interface

The purpose of the graphical user interface is to provide an abstract display and input for the user. The main class that instantiates the user interface is `SCU_GUI`. However, `SCU_GUI` also calls several other component classes. When all components are combined, they create the user interface in its entirety.

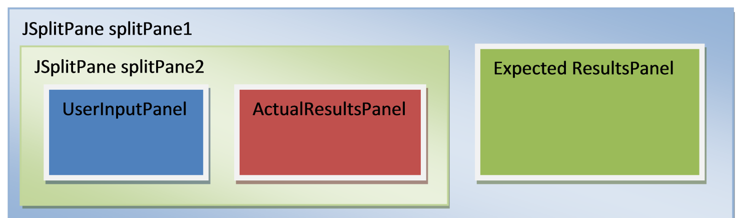
`SCU_GUI`

The purpose of the `SCU_GUI` class is to be the main hub for viewing components to be initialized and instantiated. Those components are then laid out and arranged in the `SCU_GUI` which extends `PluginTab`.

Basic Implementation Philosophies
The `SCU_GUI` first sets the size of the class to a large dimension. The reasoning for this is that the `SCU_GUI` class extends the `PluginTab` library, and is not inherently a container class. Therefore, it is necessary to set the size of the class so that all other subcomponents have a point of reference for their dimensions. This way the project could set layout standards beyond what `javax.swing` thought would be the better layout format.

The subcomponents initialized in this class are: `UserInputPanel`, `ActualResultsPanel`, and `ExpectedResultsPanel`. Each of these components are arranged into two `JSplitPanes`, set up as in **Figure 13**.

Figure 13: `SCU_GUI` conceptual layout



A "`splitPane2.setOneTouchExpandable(true)`" has been called so that the user is given the choice to hide the Expected Results panel with a single click to the arrow on the split bar.

`arrange()`
This method is of significance because this is where the main arranging of the components happens. Therefore, if a programmer wanted to edit the existing components, add content, or rearrange the components, he or she would make those modifications in this method.

`class UpdateThread`
This is not a method within the `SCU_GUI` class, rather it is another class with in the `SCU_GUI.java` file. This class basically calls the `PMCentralInterface.getStatus()` method and pulls the information from the system. The class then tells the subcomponent `ActualResultsPanel` to update its graphics and numerical displays.

`UserInputPanel`

This class holds all components and graphical user interface components needed for the user to configure the testing environment. For example, the user is able the set the error flags by check marking the `JCheckBoxes`.

`ActualResultsPanel`

This class holds all components and graphical user interface components needed to display results from the `PMCentralInterface.getStatus()`. The `SCU_GUI` class makes the `getStatus()` call and then updates all numerical and textual displays and repaints the panel.

This class also instantiates and initializes the Azimuth (`DrawTopViewAntenna` class) and Elevation (`DrawFrontViewAntenna` class) graphics. When the `updateFromStatus(Status)` method is called, data from the `Status` object is sent to the graphic classes to change the images to reflect a movement animation.

`updateFromStatus(Status)`
This method parses the `Status` object that was passed and updates all graphical display variables to the new data. Then when a refresh/repaint is called, the user will see new numbers and animations.

`ExpectedResultsPanel`

This class holds all components and graphical user interface components needed to display data that the MEU is currently aware of. Every time status is sent to the MEU, the information is pushed to this class which updates all of its display components.

`updateFromStatus(Status)`
This method parses the status object that was passed and updates all graphical display variables to the new data. Then when a refresh/repaint is called the user will see new numbers and animations.

`DrawLegend`

This class primarily returns a simple panel with color bars and labels that explain the drawn objects in the `DrawFrontViewAntenna` and `DrawTopViewAntenna` graphics.

`DrawFrontViewAntenna`

This class returns a graphic that is pulled from a source file as well as draws additional `java.awt.Shape` objects so that the end result is an image that resembles an abstract view of the gimbal antenna from the front view perspective.

`updateImage(double, double, double)`
This method gets called from outside classes that change the angle of the image itself and the negative and positive soft stops. This end result is that the method is called 120 times a second and the user sees a seamless animation of the graphic.

`DrawTopViewAntenna`

This class returns a graphic that is pulled from a source file as well as draws additional shape objects so that the end result is a image that resembles an abstract view of the gimbal antenna from the top view perspective.

`updateImage(double, double , double)`
This method gets called from outside classes that change the angle of the image itself and the negative and positive soft stops. This end result is that the method is called 120 times a second and the user sees a seamless animation of the graphic.

3 MEU Simulator

A small MEU simulator has been provided alongside the SCU simulator. The MEU simulator is a small program which can send commands over a serial port in the same format as the MEU itself would. This is useful for the purposes of testing the SCU simulator and the MEU simulator fits into our high-level architecture in precisely the same placement and manner as the MEU itself would.

3.1 Installation

Before installing the MEU simulator, the following programs should be installed:

- JDK (v1.6 or higher)
- Apache Ant (any version)
- `javax.comm` package (see **Appendix B**)

To install the MEU simulator:

1) Open a terminal at the directory containing the build.xml file and run the command "`ant meu`" without the quotes
2) This will create a file named meuSimPlugin.zip which contains a plugins directory with the MEU plugin contained in its own directory, MEUSimulator, inside the plugins directory.

3) Copy the MEUSimulator directory to the framework plugins directory
4) Run the plugin framework and use the MEU simulation

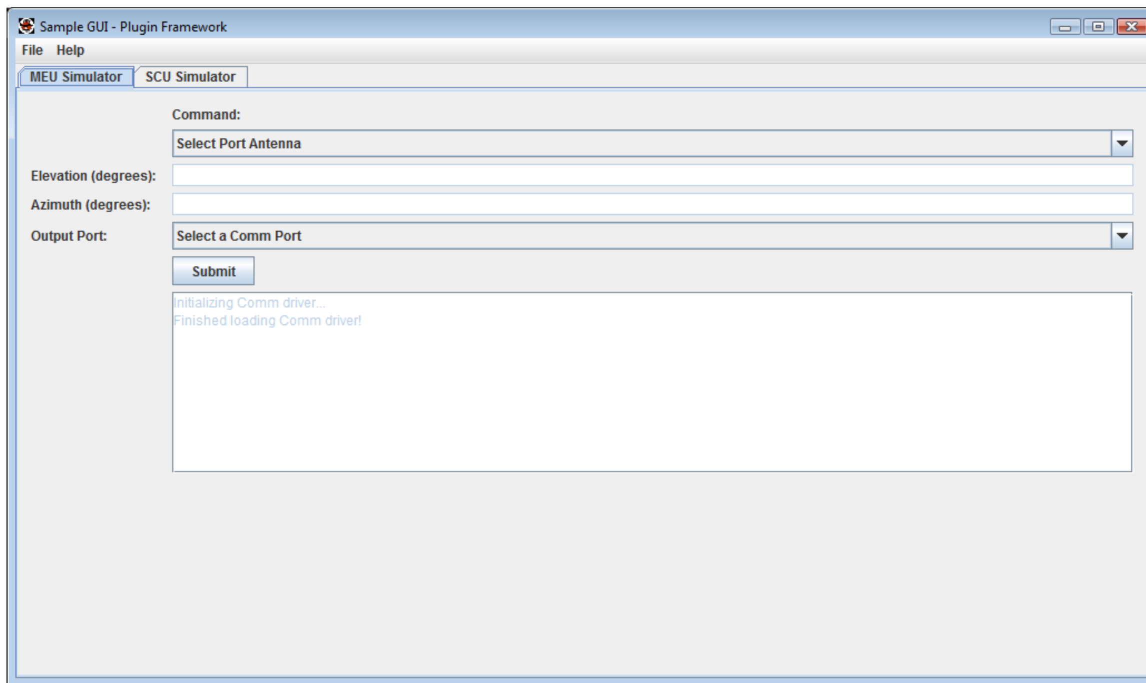
3.2 Interface

The purpose of the MEU simulator is to allow for testing of the Servo Control Unit (SCU) simulator. This plugin is able to issue commands over the serial connection, and receive the resulting status messages from the physical simulation.

Commands can be sent to the SCU simulation by selecting a mode from the uppermost drop-down box, entering Elevation and Azimuth values, if needed for the selected mode, and clicking the "Submit" button.

The "Output Port" drop-down box is used for selection of the serial port over which communication will take place.

Figure 14: The MEU Simulator



3.3 Implementation

The MEU simulation is an entirely contained unit within the `MEUSimulatorGUI` class. Most of the methods are variations of the methods found throughout various classes in the servo controller simulation. The important methods in `MEUSimulatorGUI` class are: the methods which create the byte packets for serial commands, and the `setPort(CommPortIdentifier)` method which sets the port and starts serial communications on an asynchronous thread.

A Acronym Glossary

The following is a list of acronyms used in this doucment and their meanings

ARL-M	Airborne Reconnaissance Low-Multifunction
GUI	Graphical User Interface
JDK	Java Developer Kit
MEU	Modular Electronics Unit
SCU	Servo Controler Unit

B Serial Communications Packages

The servo controller and MEU simulators are both designed to work with the javax.comm serial communications package. This package can be downloaded for free from http://java.sun.com/products/javacomm/. Setting up the `javax.comm` package is easy for *nix based machines. Simply copy the downloaded comm.jar file to the <JAVA_HOME >/lib/ext and <JDK >/jre/lib/ext directories.

Installing `javax.comm` on a Windows machine is a more involved process. To install on a windows machine,

1. Dowload the `javax.comm` package for the "Generic" platform
2. The RXTX serial drivers must be installed. The RXTX serial communications drivers can be downloaded for free at http://rxtx.qbang.org/pub/rxtx/rxtx-2.0-7pre1-i386-pc-mingw32.zip.
3. Copy the comm.jar file for the `javax.comm` package and the RXTXComm.jar, rxtxSerial.dll and rxtxParallel.dll files to the <JAVA_HOME >/lib/ext and <JDK >/jre/lib/ext directories
4. Create a new text file named javax.comm.properties that contains the single line:
   `Driver=gnu.io.RXTXCommDriver`
   in the <JAVA_HOME >/lib directory

It is possible on any machine to use solely the RXTX serial communications drivers. To do this, replace every instance of `javax.comm` in the source code with `gnu.io` and rebuild the plugins after issuing an "`ant clean`" command. Instructions for installing the RXTX communications drivers are provided on the RXTX website http://rxtx.org.