

|  |
| --- |
| APS  TMT.CTR.TEC.17.013.DRF01 |

April 13, 2017

# 

TABLE OF CONTENTS

1 Introduction 4

1.1 Audience 4

1.2 Purpose 4

1.3 Scope 4

1.3.1 Items Not Included in First Draft 4

1.4 Applicable Documents 4

1.5 Reference Documents 4

1.6 Change Record 5

1.7 Acronyms 5

2 Overview 6

2.1 System Context 6

2.1.1 Bench Layout of Stages 6

2.2 Assumptions/Constraints 7

2.3 Commitment to OSW Common Services 7

3 Conceptual Design 8

3.1 Components/Lifecycle 8

3.2 Component Architecture 8

3.3 Commands 9

3.4 Telemetry 9

3.5 Events 9

3.6 Logging 9

3.7 Configuration 9

3.8 Engineering User Interface 10

4 Assembly Design 11

4.1 Design Goal: Code Reuse 11

4.2 Assembly Class Diagram 11

4.3 Assembly Interfaces 12

4.4 Assembly Command Handling Sequence 13

5 HCD Design 15

5.1 Assembly to HCD API 15

5.1.1 Query Messages 15

5.1.2 SetupConfig Messages 16

5.2 HCD Command Handling Design 16

5.3 HCD Telemetry Design 17

6 Controller Software Design 18

7 engineering UI Design 19

7.1 Screen Mock-ups 19

8 Software Development Environment 22

8.1 Development Tools 22

8.1.1 Github 22

8.1.2 Scala IDE for Eclipse 22

8.1.3 WebStorm 22

8.1.4 Galil Controller Tools 22

8.2 Testing 22

8.2.1 Testing Tools 22

8.2.2 STIL Test Environment 22

8.3 Deployment Architecture 22

# Introduction

This document is the principle design documentation for the APS Stimulus Prototype software. A new version of this document will be delivered with each milestone of the project, as the design becomes more complete and refined.

## Audience

This document is targeted primarily towards APS team members to understand how the software was designed and what lessons were learned. This document may also be of interest to other subsystem software developers that are using or will use the OSW common services framework.

## Purpose

This document will be a key source of input to the preliminary design of the APS ICS system, particularly in the areas of CSW integration and component design with emphasis on motion control. The corresponding software development effort and technology choices will also inform similar choices for the APS ICS preliminary design.

## Scope

This document provides the design basis for the APS Stimulus Prototype based on TMT Common Services and the APS Stimulus Software requirements. This document will evolve as the prototyping exercise continues, particularly in the area of HCD/Controller design, which will become available sometime after the first draft.

### Items Not Included in First Draft

Hardware controller software design is not included in the first draft. As the team acquires experience in this area, a design will be developed.

HCD design is tentative and incomplete in the first draft. The HCD design is dependent on the hardware controller software design.

Detector software design is not in scope for this development phase of the prototype, but would be useful to include in a subsequent phase as this could drive detector assembly/HCD design for APS-ICS.

Integration with the Alarm system as defined for CSW is not in scope for this development phase of the prototype, but would be useful to include in a subsequent phase as this could drive defining alarm states for APS-ICS.

## Applicable Documents

1. [APS Stimulus Prototype API](https://docushare.tmt.org/docushare/dsweb/Get/Document-61692/StimulusPrototypeSoftwareAPI_DRF01.docx), TMT.CTR.ICD.17.006.DRF01

## Reference Documents

1. [TMT Software Development Process](https://docushare.tmt.org/docushare/dsweb/Get/Document-57862/SoftwareDevelopmentProcess%20v10_clean.docx), TMT.SFT.TEC.16.010.DFR01
2. [TMT Software Detailed Design (CSW) Common Software Subsystem](https://docushare.tmt.org/docushare/dsweb/Get/Document-58152/CSWDetailedDesign-CSWFD_REL02_signed.pdf), TMT.SFT.TEC.16.005.REL02
3. [APS Stimulus Review](https://docushare.tmt.org/docushare/dsweb/Get/Document-56800/aps_stimulus_review.pptx), TMT.CTR.PRE.16.066.DRF01
4. [Technical Document: Software Design Patterns for Device and Component Controllers](https://docushare.tmt.org/docushare/dsweb/Get/Document-57492/cc_design_patterns_REL01.pdf), TMT.INS.TEC.16.079.REL01

## Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Who | Modifications |
| *DRF01* | 3/21/17 | SM | Initial Draft |
|  |  |  |  |
|  |  |  |  |

## Acronyms

**APS** Alignment and Phasing System

**API** Application Programmer Interface

**CS** Configuration Service

**CSW** Common Software

**DRD** Design Requirements Document

**HCD** Hardware Control Daemon

**ICS** Instrument Control System

**IDE** Integrated Development Environment

**OMOA** Observing Mode Oriented Architecture

**OSW** Observatory Software

**TMT** Thirty Meter Telescope

# Overview

## System Context

The Stimulus Prototype software will control four motion control stages within the stimulus. Stand-alone software will be used to control the stimulus deformable mirror itself and to collect images from a detector.

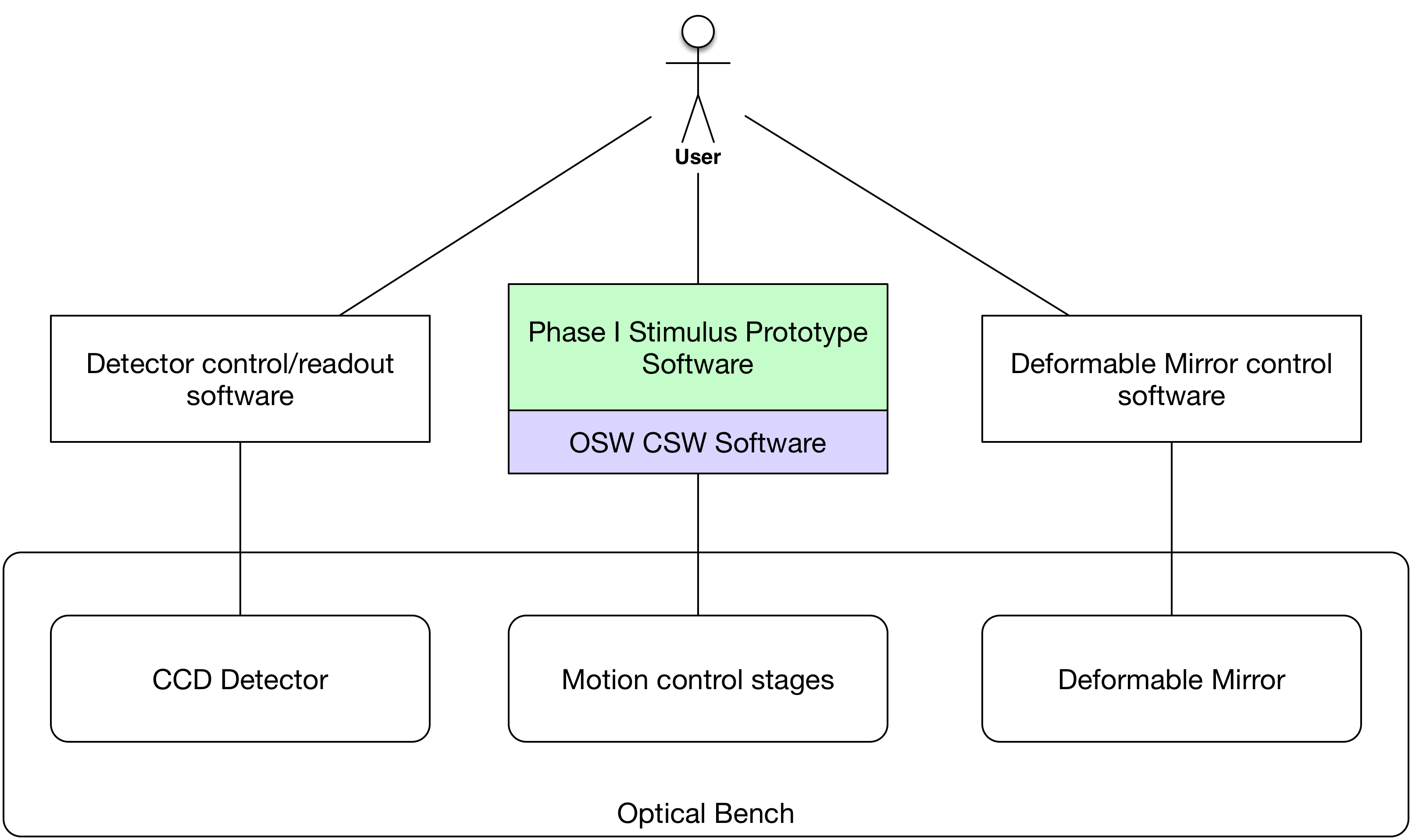


Figure - System Context

A user of the stimulus prototype will use three software systems in Phase I: the prototype software which will use CSW software libraries and follow CSW design practices, the detector control/readout software (TBD), and the deformable mirror control software provided by the deformable mirror vendor.

In a future phase it will be desirable to include detector control software into the prototype software, as this activity will be key in developing a preliminary design for ICS components related to detectors.

### Bench Layout of Stages

The following diagram shows the four stages in the context of the optical bench (not scaled) and showing the optical path.

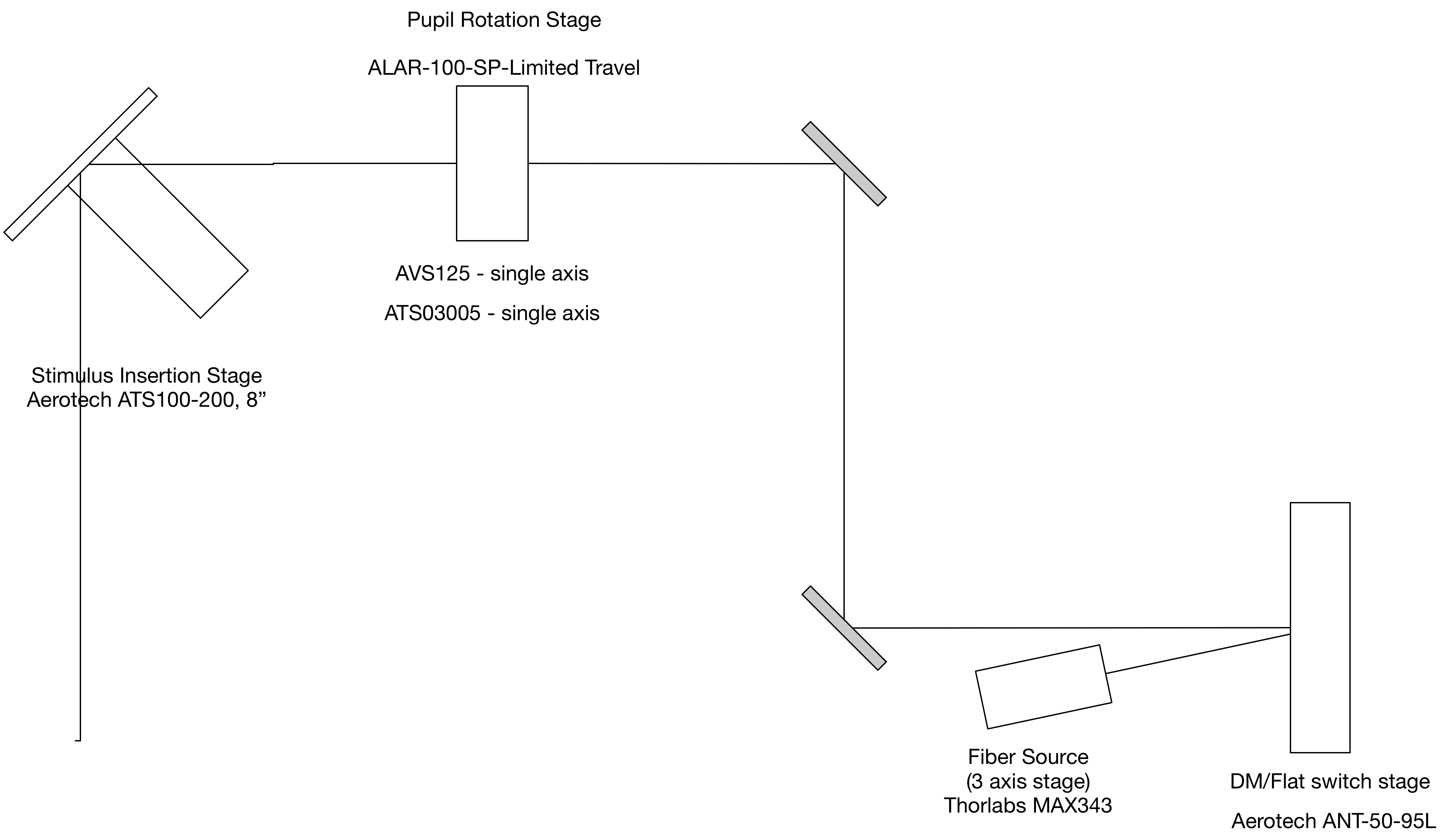


Figure - Stage Layout

The stimulus light source is a fiber source on a 3-axis Thorlabs stage. The beam is reflected off either the deformable mirror or a flat mirror, which can be switched between using the Aerotech ANT-50 stage. The beam continues to the pupil rotation stage, which is also a three-axis stage. The final stage is a two-position insertion stage, which will be used to switch between light from the telescope and light from the stimulus to the rest of the APS.

More detailed information on the bench optomechanical design can be found in RD03.

## Assumptions/Constraints

The design will make the most use possible of CSW, so that its alignment with APS-ICS needs can be evaluated and uncover any issues with the CSW design itself.

The implementation will be in the Scala programming language, so that the team may become more familiar with it, and to determine if it is a better choice than Java for APS-ICS.

User Interface technologies used for the engineering user interface will be chosen based on current OSW preferences and current market trends, potentially filling gaps in TMT knowledge of available products.

## Commitment to OSW Common Services

APS is committed to using OMOA architecture, framework and components as well as OSW common services (CSW). This prototyping will be useful in helping the team gain vital experience with CSW and the lessons learned can be used to provide feedback useful to OSW and other subsystem stakeholders.

# Conceptual Design

The ICS Prototype will use OSW common services framework and services.

## Components/Lifecycle

The software will use CSW framework components (i.e. Assemblies and HCDs) and associated lifecycle and CSW services.

## Component Architecture

The component architecture consists hardware, a controller layer, an HCD layer, an Assembly layer and a user interface layer (Figure 3).

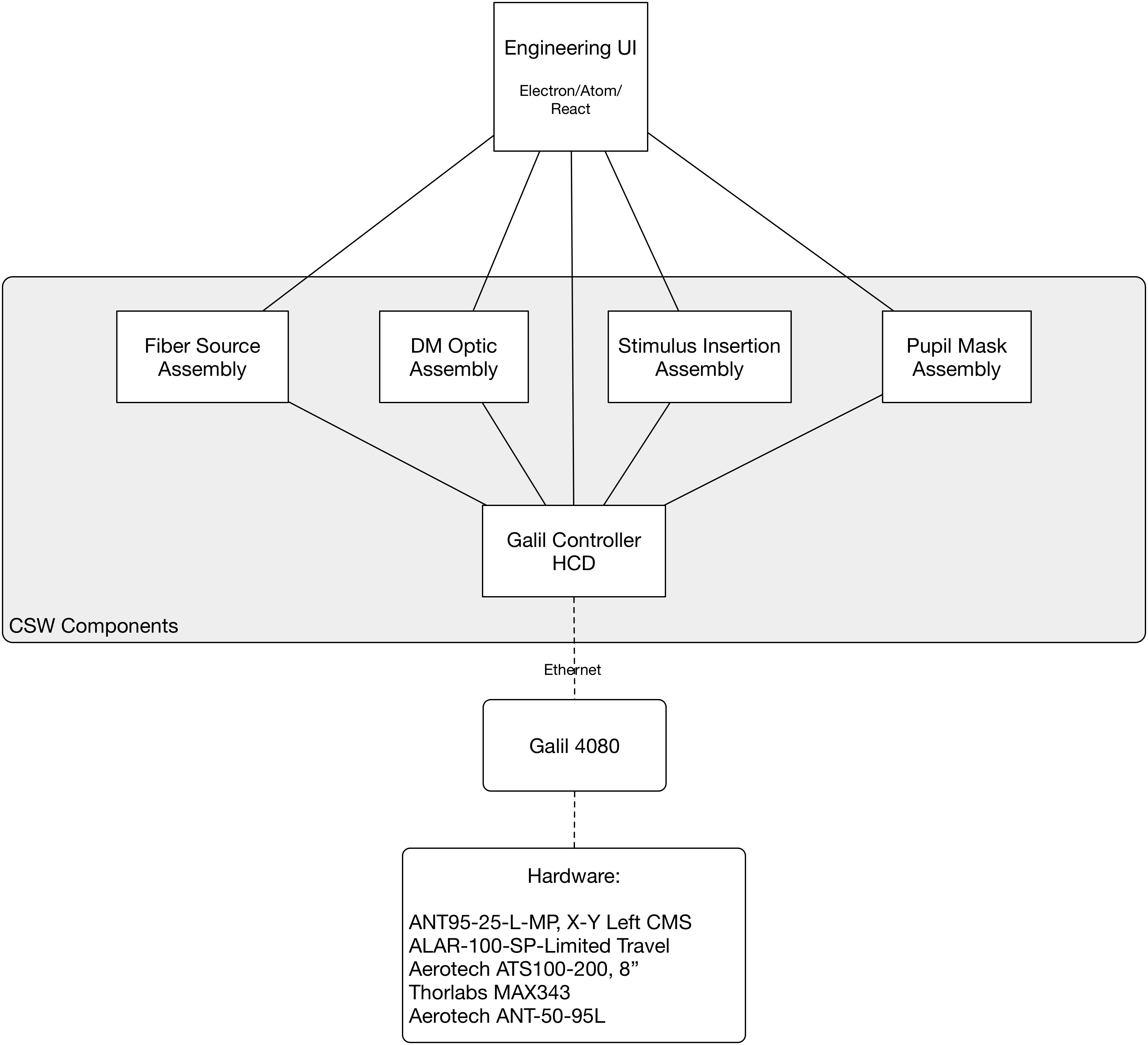


Figure - ICS Prototype Component Architecture

An 8-axis Galil controller will control the physical stages: 3 axis Fiber Source (Thorlabs Max343), a 3 axis Pupil Mask assembly, a single axis DM Optic assembly and a single axis Stimulus Insertion axis.

The Galil Controller HCD will communicate with the Galil controller over an Ethernet connection. The HCD is the sole point of communication access to the Galil controller from the rest of the system. The HCD represents the controller hardware in the CSW JVM.

The prototype will have four assemblies which are logical software representations of the user devices: Fiber Source 3-axis stage, DM Optic stage, Pupil Mask ‘composite’ 3-axis stage (a combination of three single-axis products from Areotech), and a Stimulus Insertion stage. The APS Stimulus Prototype API (AP01) is the interface to each of these user level logical devices.

The user interface will be a web-based GUI that will serve as both the means to exercise the defined API for assemblies and as an engineering UI to debug and diagnose problems. To exercise the assembly API, the UI will only communicate with the Assemblies. To debug and diagnose problems, the UI will communicate with both Assemblies and the HCD directly.

## Commands

Assemblies and HCDs will use the CSW command service for all command configurations accepted in the system. The commands that each assembly will accept are defined in (AD01).

Commands that drive hardware are transformed by the assembly into encoder units and the transformed commands are passed to the HCD.

The engineering user interface prototype will support sending commands to Assemblies and to HCDs directly.

## Telemetry

The system assemblies will produce telemetry streams as defined in (AD01). The CSW Telemetry Service will be used for all telemetry streams.

State and status messages are passed from the HCD to assemblies. These messages will be transformed into appropriate coordinates and units and published as telemetry.

The engineering user interface prototype will display telemetry streams.

## Events

By definition, CSW event publishing can only be used for information that results in actions being taken by a consuming assembly, such as a ‘follow’ scenario.

The prototype will not be using CSW events.

## Logging

The system will use the OSW Logging service to collect logging information throughout the system. Logging will include method trace logging, detailed error logging and debug logging.

A design goal will be to view logging directly from the engineering user interface. This is not currently in scope.

## Configuration

The system will use the OSW Configuration service to read and write configuration data used in the system.

Configuration will include:

* User coordinates/unit conversions to encoder units.
* Stage channel assignments
* Default stage positions
* Enumerated stage positions

## Engineering User Interface

The engineering user interface will support the running and debugging of the prototype system. The user interface will provide the ability to:

* send all API commands to assemblies
* send low level commands to HCDs
* view telemetry events
* send lifecycle commands to components
* view lifecycle state information for each component
* view error information and debug logging
* view in-progress commands and state information for commands
* cancel in-progress commands
* view and update configuration values

# Assembly Design

There are four assemblies associated with motion control in the stimulus prototype:

1. Fiber Source Assembly
2. DM Optic Assembly
3. Pupil Mask Assembly
4. Stimulus Insertion Assembly

Two of the assemblies control 3 axes each (Fiber Source and Pupil Mask) and accept commands to position each of their axes relative to their current positions. The other two assemblies (DM Optic and Stimulus Insertion) accept commands to position a single stage to one of two defined positions.

Much of the assembly design borrows from the CSW vertical slice implementation and RD04 and adapts the design to APS needs as well as refactoring for code reuse, etc.

## Design Goal: Code Reuse

It is a design goal to reuse code as much as possible. Each assembly will have the following similar or identical duties:

1. Registering and maintaining CSW service references
2. Maintaining reference to an HCD
3. Handling incoming external commands
4. Handling incoming telemetry from the HCD
5. Maintaining state
6. Converting to/from scientific units/coordinates from/to controller units
7. Validating incoming external commands
8. Publishing Telemetry

The prototyping exercise will explore how code reuse can be applied to the four assemblies in the design. This design document will be updated as new designs achieving this goal are realized.

### Reusable Assemblies

The primary reuse area for the prototype will be to create reusable assemblies. The design goal is to avoid coding every assembly in the system individually but instead to create a small number of assembly classes that can be used in multiple areas through configuration.

Consequences of the design:

1. Conversion algorithms in the vertical slice are specific to a config key. If we want to create only a few generic assembly classes and use configuration to reuse.
2. Others TBD.

## Assembly Class Diagram

The StageAssembly is the Top-Level Actor for the Assembly. It is the superclass for the MultiAxisAssembly which will be used for the 3 axis stages: FiberSource and Pupil, and the superclass of EnumPosAssembly, which will be used for the discrete position stages: DmOptic and StimulusInsertion (Figure 3).

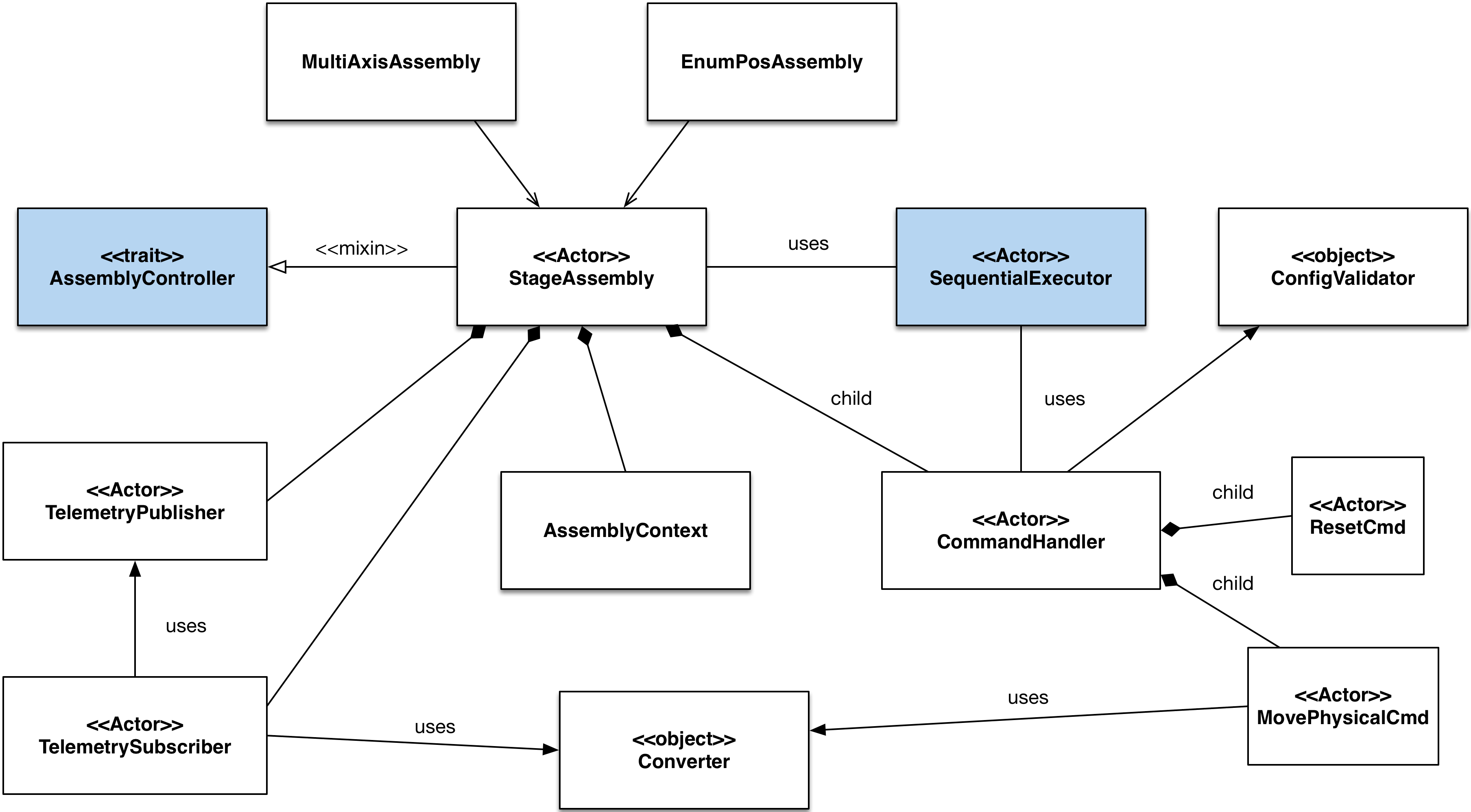


Figure - Assembly Class Diagram

The StageAssembly uses the AssemblyController mixin, which enables submitted setupConfigs to be handled by the assembly. The stage assembly must override the setup() method of the AssemblyController to handle setup configs. It is the responsibility of the StageAssembly to implement command validation, returning a validation list from the setup method. This is accomplished using the ConfigValidator object, which will have the responsibility of validating setup configs. The AssemblyContext contains all the configuration definitions and utilities required to create and send setup configurations to the assembly.

The AssemblyController and SequentialExecutor are provided by CSW and are indicated as such being rendered in the diagram in blue.

The SequentialExecutor is used so that configurations are executed one at a time for an assembly. Users of SequentialExecutor must supply a CommandHandler that handles each command as it is sequenced.

The CommandHandler delegates handling each command to a child Actor named for each command defined in (Table 1). Each of these Actors are responsible for converting commands into HCD units and coordinates using the Converter object and managing the asynchronous command and response from the HCD.

The TelemetrySubscriber listens for telemetry from the HCD and uses the Converter object to convert stage encoder units and coordinates to user units and coordinates and publishes these using the TelemetryPublisher.

## Assembly Interfaces

### ICS API Commands

Command configurations for each assembly are described in the ICD document (AP01). In this section, the set of input command configurations are defined, and the corresponding HCD command keys are listed.

|  |  |  |  |
| --- | --- | --- | --- |
| Assembly API method | Assembly Config Key | Accepted By | HCD Command Key |
| reset() | reset | All Assemblies | axisHome |
| select() | select | Discrete Position Assemblies | axisPosition |
| positionStage() | stagePosition | All Assemblies | axisPosition |
| setSelectionsPoints() | stageSelections | Discrete Position Assemblies | N/A |
| offsetStimulus(), offsetPupil() | offset | 3-Axis Assemblies | axisOffset |
| positionStimulus(), positionPupil() | position | 3-Axis Assemblies | axisPosition |
| setReferencePoints() | stageReference | 3-Axis Assemblies | N/A |

Table – ICS Assembly Commands

The ‘reset’ command is shared by all assemblies. For DiscretePositionAssembly types, the axisHomeCK is sent to the associated HCD axis for that assembly. For MultiAxisAssembly types, the axisHomeCK is sent to only those axes that are specified in the command.

The ‘select’ command only applies to DiscretePositionAssembly types. The assembly uses the enum position in its loaded configuration to determine the appropriate motion command and sends axisPositionCK to the associated HCD axis for the assembly.

The stagePosition command is shared by all assemblies. For DiscretePositionAssembly types, the axisPositionCK command is sent to the HCD axis for that assembly. For MultiAxisAssembly types, the axisPositionCK command is sent to each of the axes (or a setup config containing 3 axisPositionCK commands, one for each axis is sent).

The ‘stageSelections’ command only applies to DiscretePositionAssembly types. The command configuration data for the enumerated positions is stored in the assembly configuration by the assembly, using the configuration service.

The ‘offset’ command only applies to MultiAxisAssembly types. For each axis, offsets are converted to encoder units commands are sent to the HCD axis using the axisOffsetCK command.

The ‘position’ command only applies to MultiAxisAssembly types. For each axis, positions are converted to encoder units and are sent to the HCD axis using the axisPositionCK command.

The ‘stageReference’ command only applies to MultiAxisAssembly types. The command configuration data for the reference points for each axis is stored in the assembly configuration by the assembly, using the configuration service.

### Standard Assembly Commands

The prototype will also implement standard assembly commands as defined by RD04.

|  |  |  |
| --- | --- | --- |
| Command Configuration Key | Accepted By | HCD Command |
| init | All | TBD |
| datum | All | TBD |
| stop | All | TBD |
| debug | All | TBD |

Table – Standard Assembly Commands

## Assembly Command Handling Sequence

This section describes the sequence followed when an Assembly receives a command, i.e. a submit configuration.

Assembly received commands are executed sequentially. Any API command will be accepted when an Assembly is in the Ready state.

In the case where the HCD does not respond to a command, the assembly will return an appropriate error response.

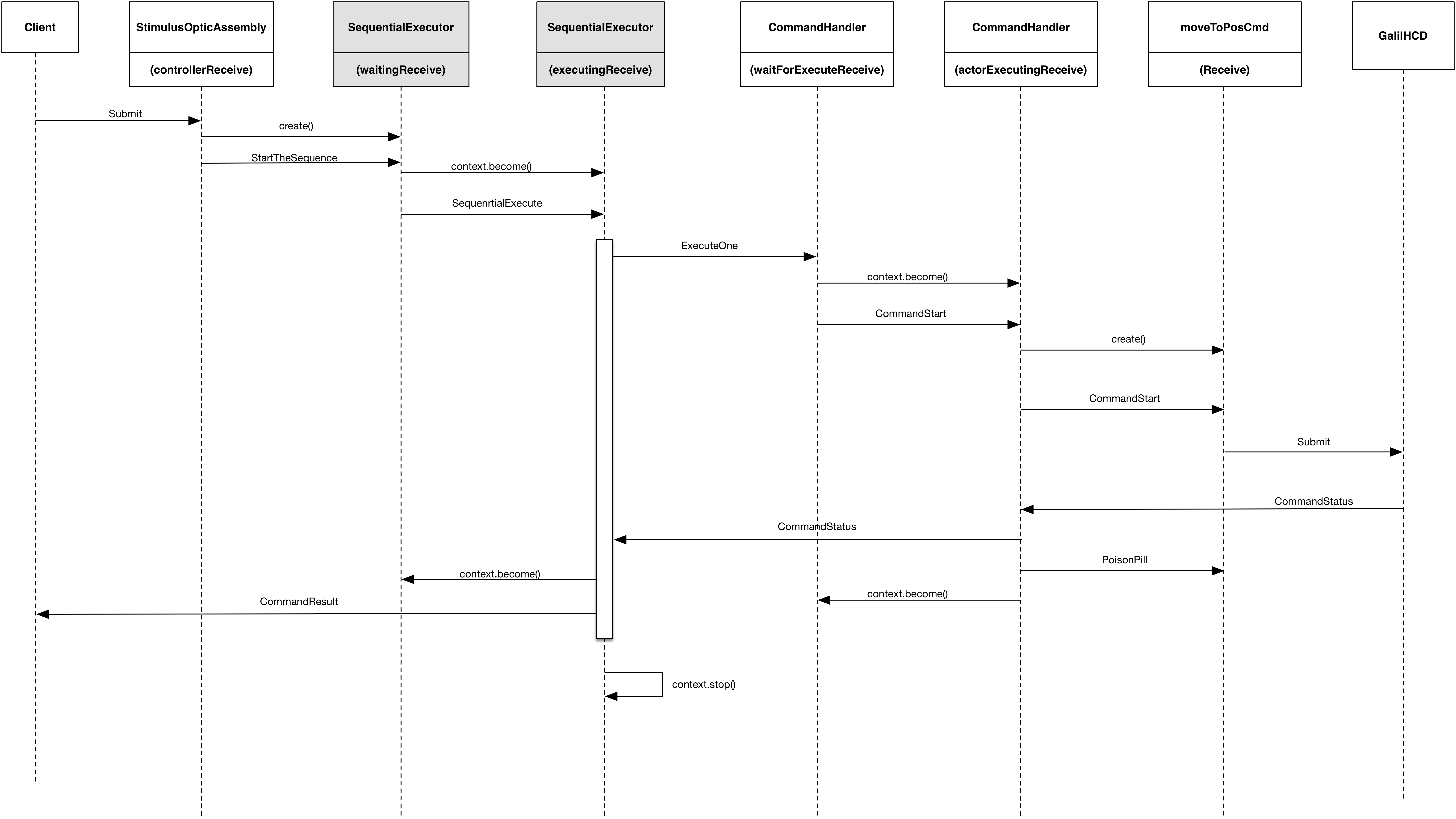


Figure – Assembly Actor Message Sequence for Single Submit Config

When an assembly receives a Submit configuration, it creates an instance of a SequentialExecutor and sends a command to start the sequence of processing the submit (Figure 5). The Sequential Executor iterates though each config in the submit and for each config:

* Sends a command to the Command Handler, which directs the config to the appropriate command actor – such as moveToPosCmd for a moveToPos command.
* The command actor sends a submit command to the HCD. When complete, the HCD sends a Command Status message to the CommandHandler, which stops the command actor, sends the Command Status back to the SequentialExectutor and returns to its ‘wait for Execute receive’ state.
* The SequentialExecutor creates a CommandResult and sends this back to the originator of the submit configuration.

When all iterations are complete, the SequentialExecutor stops itself.

## Commanding Assemblies from an External Client

The prototype will explore means to command assemblies from an external client such as will be necessary for APS-PEAS or an engineering user interface.

In the diagram below, two applications SingleAxisApp and SingleAxisClient run on separate JVMs, and SingleAxisClient sends commands to assemblies in the SingleAxisApp using the SingleAxisComponentHelper (Figure 6). The SingleAxisComponentHelper provides helper methods to send SetupConfigs to Assemblies and contains all the configKey and setupConfigs that the assembly will accept.

The name SingleAxisComponentHelper is used for the SingleAxisAssembly. A different assembly would have a different name and a different help name.

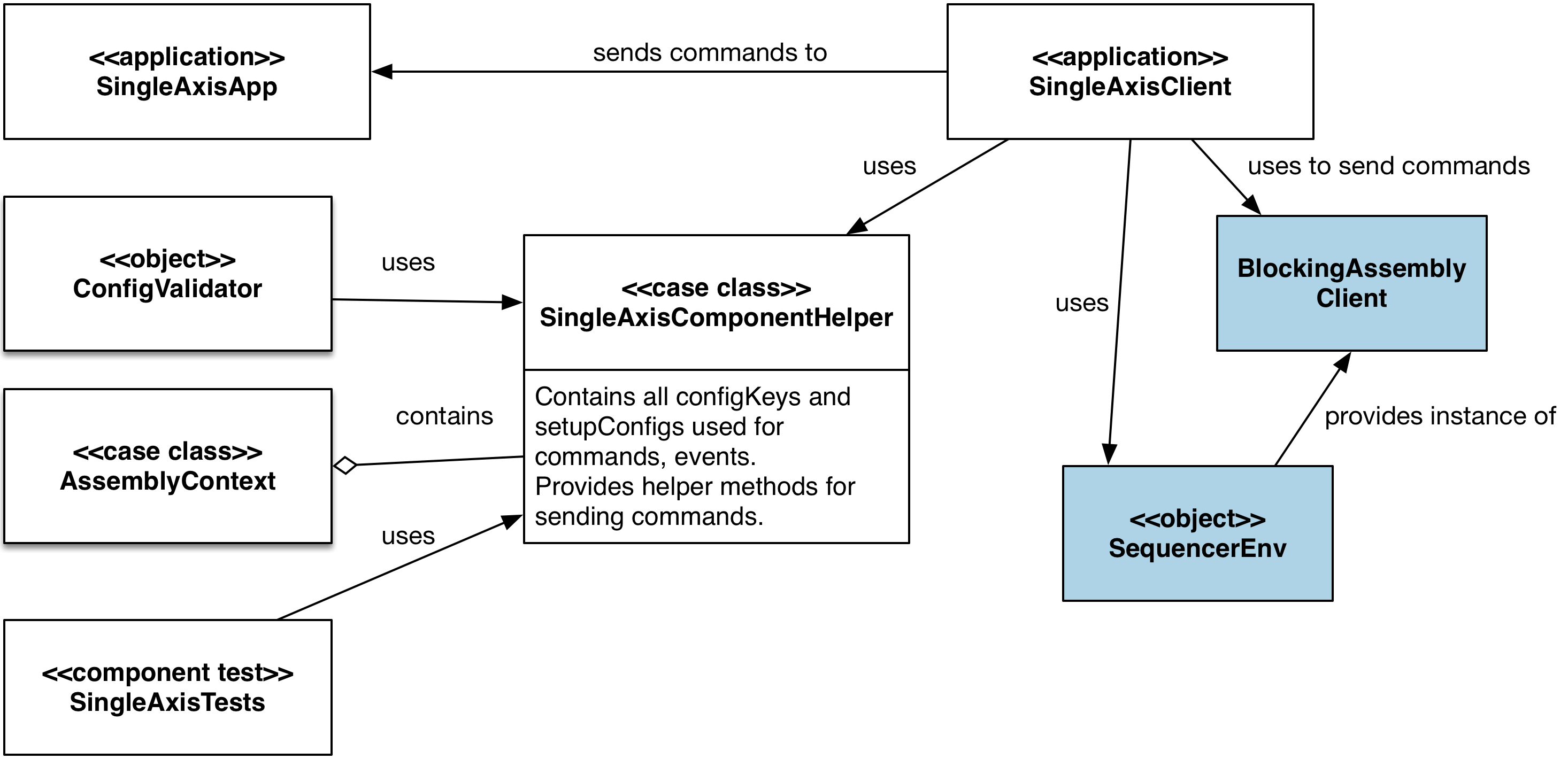


Figure - Commanding Assemblies from a Client

The utility of having the SingleAxisComponentHelper class is that the configKeys and setupConfigs for an assembly can be maintained in one place, to be used by a client program, automated component tests and by the ConfigValidator.

The AssemblyContext contains a reference to the SingleAxisComponentHelper, but a SingleAxisComponentHelper constructor only requires an assembly component prefix, which is the way a SingleAxisClient or component test would create it.

The SingleAxisClient uses CSW provided objects to send commands to assemblies, in the prototype the SequencerEnv is used to provide and instance of a BlockingAssemblyClient that the SingleAxisClient uses to send commands. It is not clear if the classes and objects provided by CSW are being supported as they are not included in the programmers manual (ref?).

## Assembly Configuration Design

TBD – will complete in a future draft

## Assembly State

Assembly state for the prototype follows standards described in RD04. The state of an assembly is a composite of a command state (uninitialized, ready, busy, error) and a move state (unindexed, indexing, indexed, moving). The prototype will also implement a selection state for assemblies that move to discrete locations.

The prototype code will not implement continuous command states or interruption of commands with other commands or cancelations that are provided for in RD04. These will be added at a later point if deemed necessary.

Assembly state is updated, used and published in the vertical slice implementation of CSW.

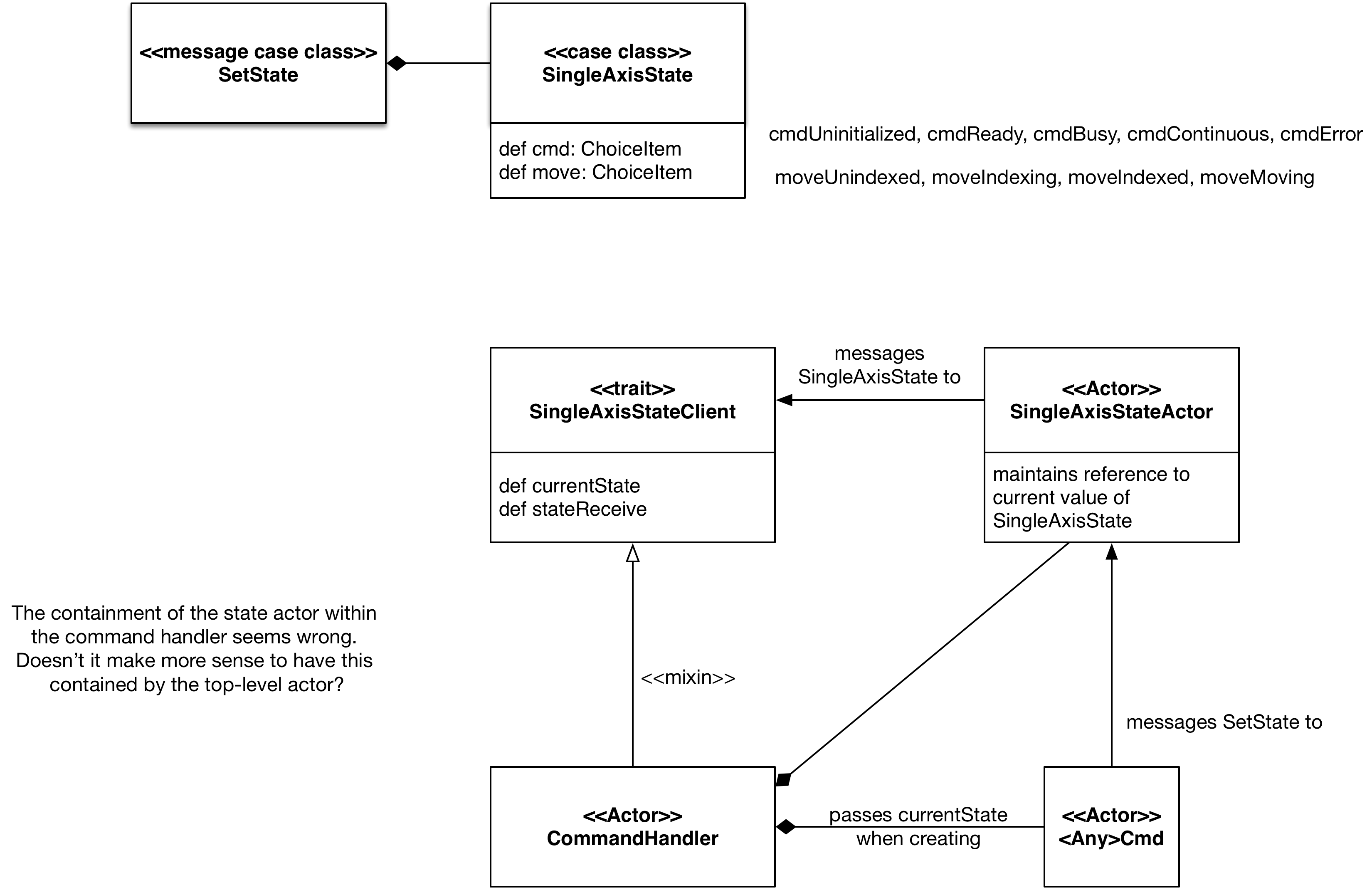


Figure - State Related Class Diagram

Figure 7 describes the classes and relationships related to assembly state. These parallel the vertical slice implementation provided with CSW. The CommandHandler takes each command and determines which command actor to create, passing in the currentState, which is an instance of SingleAxisState (Figure 8). Each command actor has logic to determine, given the currentState if the command can be executed or not. If the command is executed, the command actor sends a SetState message with the state changes appropriate to the command to the SingleAxisStateActor, that maintains the current SingleAxisState value and each time the state changes, publishes a the SingleAxisState on the Akka internal event bus. Each subcomponent actor that needs the current state can do so by mixing in the mixin trait: SingleAxisStateClient, which subscribes to the internal Akka event bus and provides the composite actor with a function returning the current state.

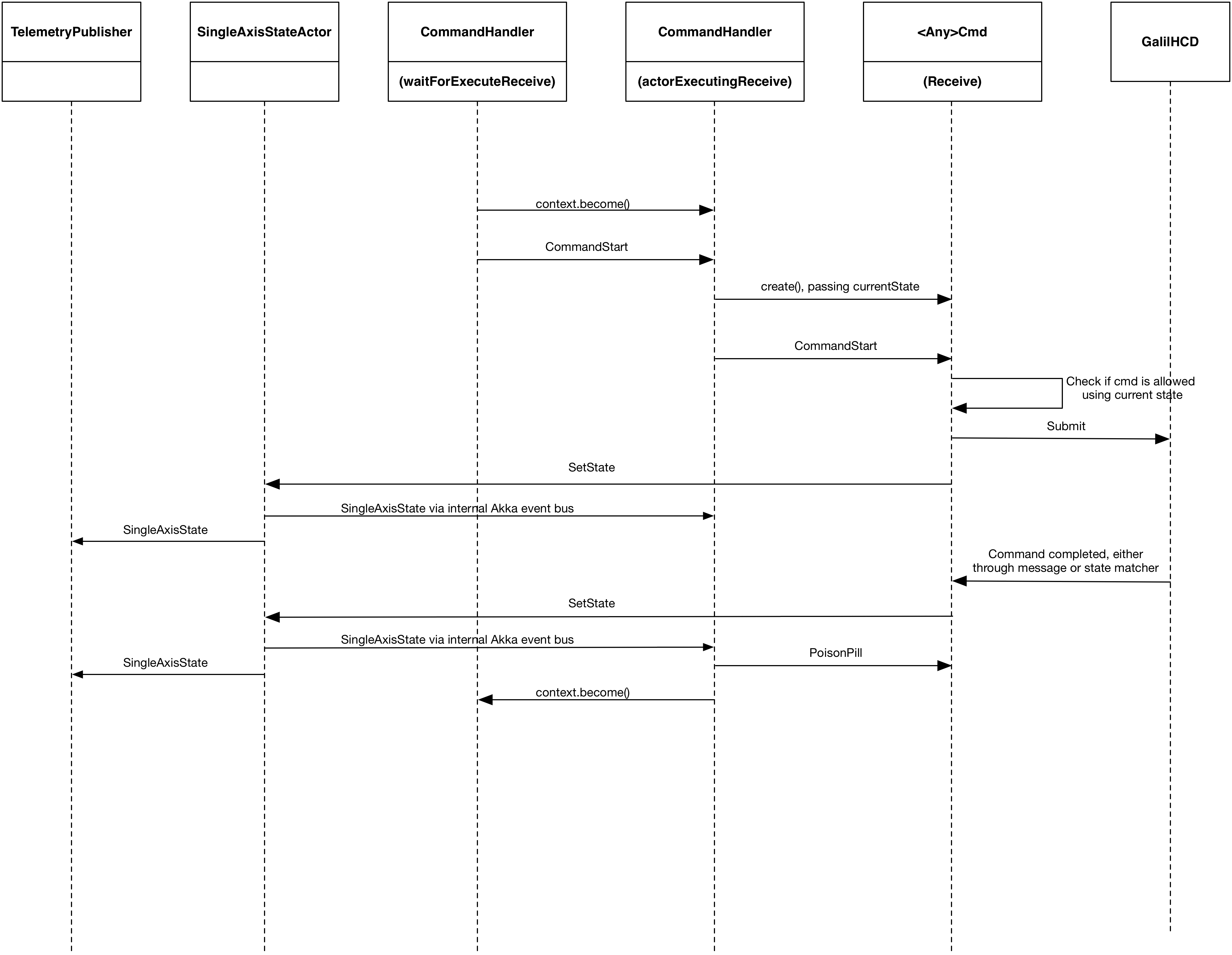


Figure - Assembly State Messaging Sequence

In the vertical slice implementation, each command actor updates the state actor with both new command and move state changes. Upon receipt of this message, the state actor publishes the assembly state using the akka internal event bus. Any assembly subcomponent actor mixing in the <assemblyName>StateClient mixin will have the current state of the assembly in a var: currentState.

In the ICS prototype, we also wanted the state to be published to the telemetry service. For this, the telemetry publisher actor reference is passed to the constructor of the state actor. Upon receipt of state changes, the state actor also sends a SingleAxisState message to the Telemetry Publisher.

## Assembly Telemetry Design

Telemetry for the ICS prototype will consist of motion control position related telemetry and assembly/stage related state information.

One component is responsible for publishing using the telemetry service: the SingleAxisPublisher. Assembly state information is published directly by the SingleAxisPublisher when it receives the SingleAxisState message.

Motion control position related telemetry is collected and assembled by the TelemetryGenerator, which can be directed to generate telemetry values at operational and diagnostic rates.

### Telemetry Generator

The TelemetryGenerator can receive the following messages:

**OperationsState**: Puts the generator into operations mode, only generates telemetry every xxx ms. <Who sends this?>

**DiagnosticState**: Puts the generator into diagnostic mode, generates telemetry for every CurrentState message. <Who sends this?>

**CurrentState**: Sends an AxisStateUpdate message to the SingleAxisPublisher at a rate determined by the operations/diagnostic state. <Who sends this?>

The vertical slice accomplishes diagnostic vs operational states by implementing two receive partial functions: one for operational receive and one for diagnostic receive. This results in much duplicate boilerplate, which is being used to avoid having a var for the HCD actor ref and the current state message counter, used to throttle the publishing in operations mode.

This is the common pattern for implementing state within an Actor. But there should be some way to avoid all the boilerplate.

### External Client Assembly Telemetry Subscription

As per CSW design, this is accomplished by registering a callback with the telemetry service using the telemetry service subscribe function. The callback is called for every StatusEvent received which is passed to the callback function.

# HCD Design

The prototype software design contains only a single HCD, the GalilHCD which controls all communication with the Galil controller. The GalilHCD is an actor that uses the mixin trait HcdController, which enables the handling of SetupConfig commands and inherits from PublisherActor, which enables publishing from the HCD.

Both PublisherActor and HcdController are CSW classes provided by OSW.

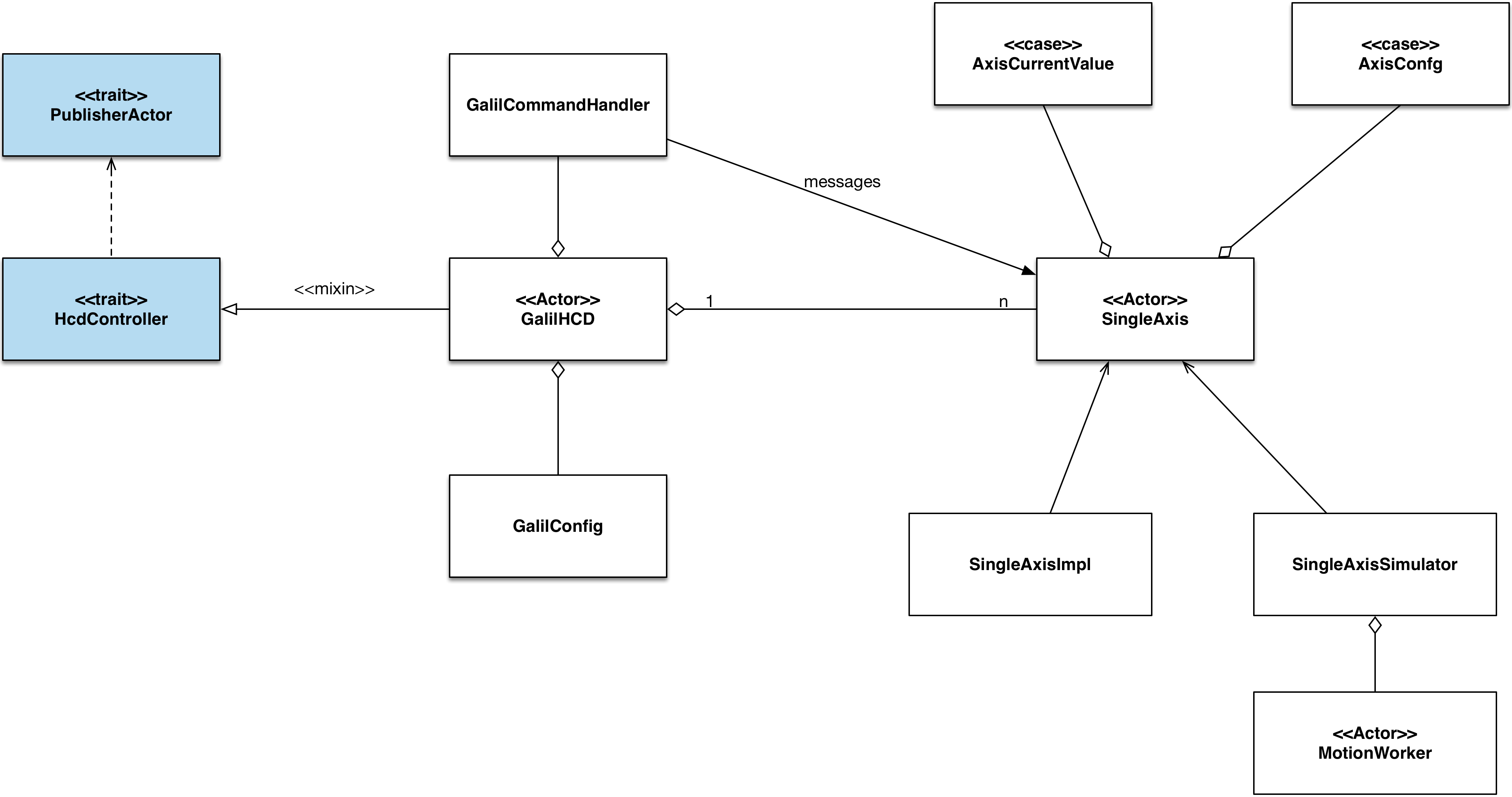


Figure - HCD class design

On startup, the GalilHCD retrieves the configuration for each axis in the Galil controller, and for each axis creates a SingleAxis actor. A SingleAxis actor can be either an actor that communicates with the Galil controller, or a simulator (SingleAxisSimulator). All actor references for SingleAxes are stored in the GalilConfig.

As per the contract defined in HcdController, incoming SetupConfig commands are processed using the process() method. The GalilHcd overrides this method and delegates all SetupConfig processing to the GalilCommandHandler.

The GalilCommandHandler uses the GalilConfig to determine the SingleAxis actor to send incoming SetupConfigs to.

## Assembly to HCD API

### Query Messages

The HCD should expose as many low-level commands/queries from the Galil as possible to facilitate debugging. These will be described in a future draft of this document, as the team gains more experience with the controller.

|  |  |
| --- | --- |
| Message | Description |
| getAxisCurrentValue(axisName) | Returns current value for a named axis |
| getAllAxesCurrentValues() | Returns all axes current values |
| getAxisConfig(axisName) | Returns the configuration values for a named axis |
| getAllAxesConfig() | Returns the configuration values for all axes |

### SetupConfig Messages

The following messages are sufficient to support the Assembly APIs.

|  |  |
| --- | --- |
| Setup Config Key | Description |
| axisPositionCK | Positions an axis to a specified position |
| axisOffsetCK | Offsets an axis a specified amount from its current position |
| axisHomeCK | Homes an axis |
| axisCancelCK | Cancels the current axis command |

## HCD Command Handling Design

The following diagram (Figure 7) shows a multi-axis HCD controller design based on the CSW vertical slice. Commands are delegated to the appropriate axis and a MotionWorker is used to simulate the controller axis. Iterating at regular intervals, encoder ‘ticks’ are used to simulate the encoder values, which are messaged back though the actor calling chain using AxisUpdate messages.

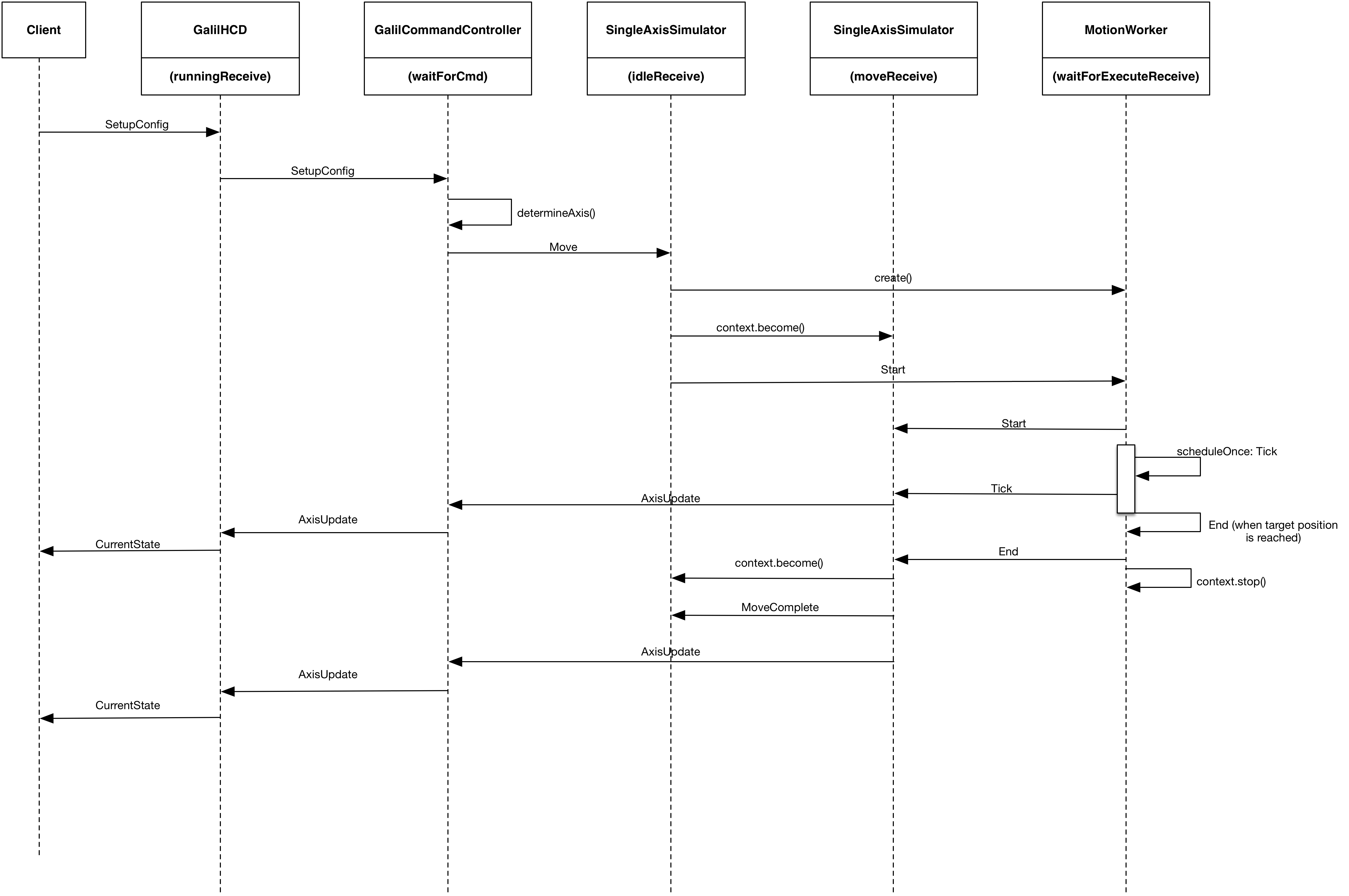


Figure - HCD (Simulator) command messages

The vertical slice uses a demand matcher at the Assembly to determine if a move command has completed, and thus only uses CurrentState messages to tell its client that the command has completed. The prototype might instead be designed to return CurrentState as telemetry, but also return a CommandState to tell the client it is done. The tradeoffs of these approaches is still being evaluated.

# Controller Software Design

TBD – this section will be developed in future drafts.

# engineering UI Design

The engineering user interface will be implemented using the React framework and Element framework for rendering HTML user interfaces as desktop applications. The React framework will be used because it is a popular framework not yet evaluated by TMT and Element is under consideration by OSW for rendering web applications on the desktop.

The design of the engineering UI will likely evolve as more is learned about the Galil controller. The following screen mockups capture the requirements known currently.

## Screen Mock-ups

The Dashboard screen allows user to see at glance the states of all components, active commands and current events (Figure 5).



Figure - Engineering UI Dashboard

The dashboard screen allows the user to stop or restart any component, cancel an in-progress command, view applicable log files and errors, and set telemetry mode between debug and normal modes.

The Assembly Control Console screen enables the user to construct commands each assembly, send the commands and view telemetry values returned from the assembly (Figure 6).

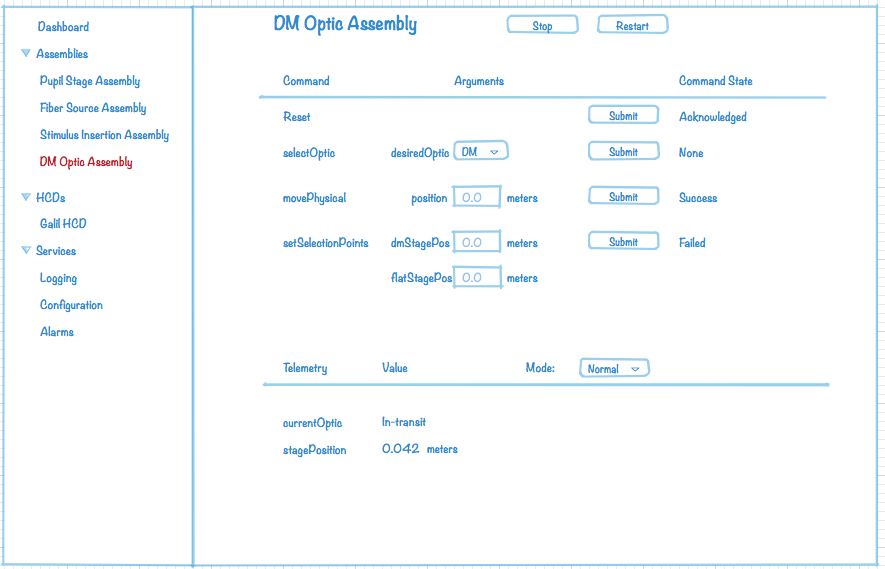


Figure - Assembly Control Console Screen

A low-level HCD command screen will also be included in the engineering UI. Details for this screen will be developed once the requirements for low level debugging of the Galil controller HCD are better understood by the team.

The Assembly/HCD configuration screen allows the user to view all configuration values for an assembly and for its associated HCD axes (Figure 7). An Edit link enables the user to change any value in the configuration database (screen not shown).

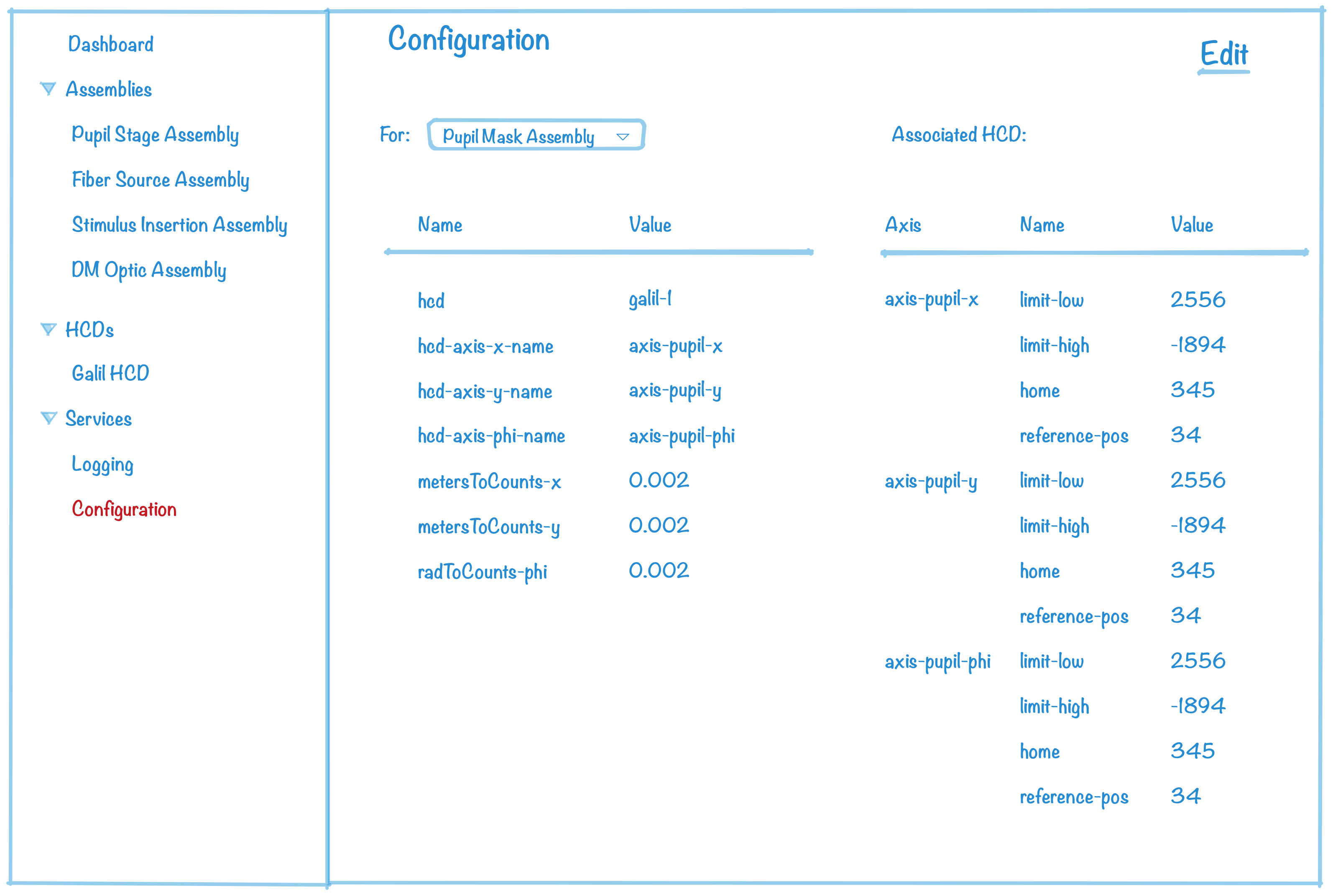


Figure - Assembly/HCD Configuration

# Software Development Environment

The project will make use of the STIL provided software development environment, including source repositories and build/test environments.

## Goals

The development environment should reflect, to the greatest extent possible, the STIL build and test environment tools and workflows, so that during the APS development phase git pull requests and subsequent build/test in the STIL environment will give the same result as in the local development environment.

The development environment should support both running with simulators and running with actual hardware.

Unit tests and component tests will be fully exercised and tested against code coverage tools being used by STIL.

## Environment Overview

The environment in Figure 14 fulfills the goals of section 8.1.

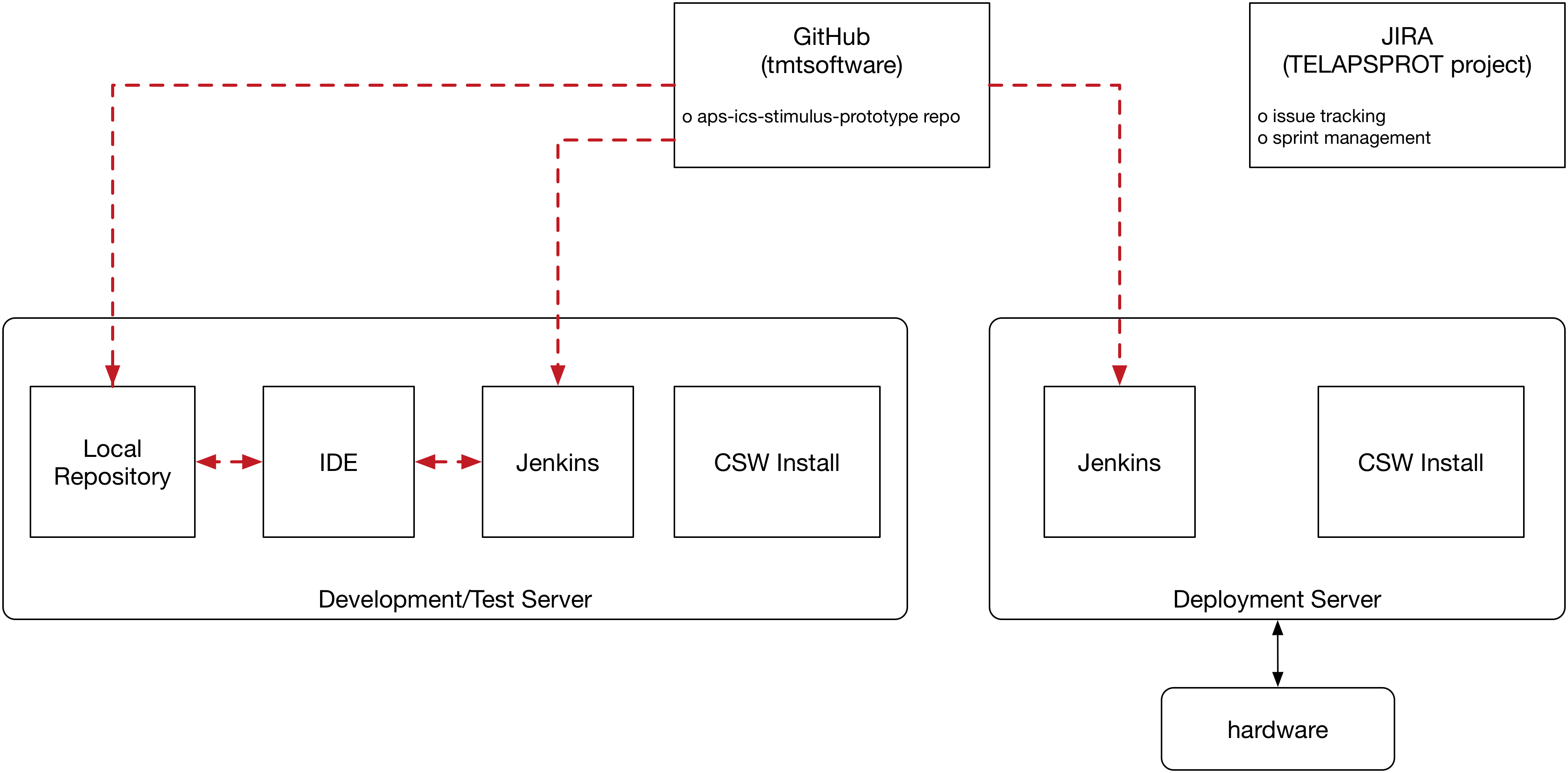


Figure - ICS Stimulus Prototype Development Environment

## Development Tools

### Jenkins

Linux:

sudo service jenkins [start | stop | restart]

For the mac:

sudo launchctl unload /Library/LaunchDaemons/org.jenkins-ci.plist

sudo launchctl load /Library/LaunchDaemons/org.jenkins-ci.plist

### Github

The project will be using github as its source control tool. STIL does not appear to have github set up for subsystem development yet, so the project is currently in the aps-stimulus-prototype repository owned by github user smichael1 (<https://github.com/smichael1>).

The project will use github as its source control repository. There will be three repositories:

* **aps-single-axis-prototype** – prototype code for the preliminary single stage prototype delivery
* **aps-ics-stimulus-prototype** – code for the full set of assemblies/hcd’s for the ICS stimulus prototype
* **aps-ics-engui-prototype** – the engineering GUI prototype code

### Scala IDE for Eclipse

Development for the prototype will use the Scala language, and the Scala IDE for Eclipse will be used as the development environment tool for assemblies and HCDs.

### WebStorm

For javascript based user interface development using the React framework, the WebStorm javascript IDE will be used.

### Galil Controller Tools

TBD – this section will be developed in future drafts.

## 

## Testing

### Testing Tools

It is a goal of the prototyping exercise that the system implements Scala unit and component testing as directed by the TMT Software Development Process (RD01). This is useful to the APS team in that it provides experience and can refine schedule estimates, and is useful to CSW in providing feedback on tools.

It will be a useful exercise to experiment with user interface testing tools, as these will be needed by other groups at TMT. This is out of scope for this prototyping phase.

### STIL Test Environment

It is a goal of the prototyping exercise that the system is deployed at the STIL Build and Test Environment (BTE), so as to gain experience and provide feedback. The achievement of this goal will be dependent on BTE functional availability and APS resource availability.

## Deployment Architecture

TBD – this section will be developed in future drafts.

# Appendix A – Single Axis Assembly Prototype

The single axis assembly prototype is a subproject to learn the CSW framework using a simple assembly for a single axis. The following notes are transitional and will ultimately be applied to the design of the ICS prototype.

## Axis State Machine

Indexed – you know where the axis is to a high degree of accuracy. For our purposes, is indexing the same as ‘Homing’?

Moving – one or more axes in an assembly are currently moving

# Appendix B – CSW Lessons Learned

## Changes Requested

Need to add ‘radians’ to UnitsOfMeasure – will use ‘degrees’ until addition is made by CSW

Packaging: CSW does not use inverse domain prefixes. I have never seen this practice.

Camelcase: HCD, CK, SC are abbreviations. I learned to put abbreviations in camelcase such as Hcd, Ck, Sc. This helps readability when other words and abbreviations are concatenated with the abbreviation: e.g. commandHCDSC is easier to read as commandHcdSc.

Reformatting of source code does not play very well with IDEs. Maybe this should be an optional sbt goal.

Message sending log output from actors does not include the class name of the actor sending/receiving, but something more cryptic. We should find a way to handle this better.

It is not clear from the Location Service “Registered” debug output what name should be used to access the assembly component in an external client:

Registered Akka singleAxisAssembly-Container-akka at akka://singleAxisAssembly-system@10.210.211.94:53569/user/singleAxisAssembly

In the case above, one must use the name “singleAxis” as the input arg to the resolveAssembly function of the SequencerEnv object. The problem here is that the input arg is called “the name of the assembly”. The ‘name’ parameter for the assembly in the conf file is singleAxisAssembly.

<< look up the above in the programmers manual. Perhaps there is a discussion there>>

Command prefixes (init, position, stop, etc) are hard-coded into Demo.scala and test cases. This is also true of the setup config Item keys, types and units. In fact I got into trouble using different objects to define the same configurations (e.g. SetupConfig, config keys). The AssemblyContext contains these and Demo.scala also contains them. The information for which keys, and setupConfigs are used should be in a single object. I designed a solution for this in section 4.5.

The AssemblyWrapper waitingForResult Receive function should at least print out the CommandResult overall command status value along with the “Received final” message.

So far I cannot get the postLastEvents option of the Telemetry Service subscribe to work. I get all the events generated after the client subscribes, but not the previous/original value.