W. M. KECK OBSERVATORY

ASTRA
FDL Sequencer

Author	Modified	Notes
Doug Morrison	10/21/08	ASTRA FDL Sequencer Design Document

1.0 Introduction

The Delay Lines are designed to adjust the optical path length of the light from the two Keck telescopes in order to maintain fringe alignment at the FATCAT detectors. This is done by providing periodic updates to the Delay Lines based on the projected path length as time passes for a specific target. The current V2 FDL Sequencer is responsible for calculating these new positions and sending the offsets to the Delay Lines ever few seconds. The sequencer uses the Star Catalog to obtain information on the currently selected target, and the MSC library routines to calculate the required offset based on the telescopes location (latitude, longitude, altitude) and projected time. Each pair of Delay Lines has its own instance of the FDL Sequencer, all of which are initialized and controlled by the higher level V2 Sequencer.

The current implementation of the FDL Sequencer restricts observing to a single target. Although this is satisfactory for every mode of observing that has been performed up to this point, ASTRA's Dual Field Phase-Referencing and Astrometry modes will require multiple star targets to be processed simultaneously.

This document will describe the proposed design plan for the development of an ASTRA FDL Sequencer (Target Generator) based on the original V2 implementation. The following sections will detail the technical requirements, interfaces, and risks associated with this task.

2.0 Overview

The Dual Field Phase-Referencing and Astrometry capabilities of ASTRA will use two targets; a bright primary source to act as calibrator and stabilizer for the IF system, and the dim science target to be observed. Since ASTRA will be observing two targets at once we will need to make use of both the primary and secondary FDLs, unlike V2 where we only utilize the primary beamtrain.

During observing, the K-band delay lines will need to track the target as it travels through the sky. The FDL Sequencer is designed to calculate the required positions of the FDLs and send these targets to the Delay Lines. Since the target positions are associated with a specific timestamp and need to arrive at the FDL early enough to be processed into the track offset, the FDL sequencer actually calculates positions for times that are a second or two in the future. These positions will be buffered by the FDLs until the timestamp is current and ready to be incorporated into the cart position.

2.1 Requirements and Design

Ideally we would like to reuse the existing target generation routines and algorithms as they exist in the V2 FDL Sequencer; this will reduce the amount of work that goes into developing and testing the ASTRA FDL Sequencer. In order to accomplish this we will need to modify how the target generation code obtains the target positions in order to associate different targets with each set of Delay Lines.

A simple solution to this problem is to have the higher level Observing Sequencer determine the appropriate target for each FDL Sequencer. The Observing Sequencer would then provide each FDL sequencer with the appropriate target information in a method call that will also initiate the generation process. This design will allow us to decouple the target generation routine from the star catalog and other components, creating a more compact and independent version of the FDL Sequencer.

2.2 Language Choice and Dependencies

Originally, Python was explored as the design language for the FDL sequencer based on its ease of use and increasing prevalence in the IF subsystems. However after further investigation it was decided that the risks encountered (such as loss of dynamic reconfiguration and reduced ability to utilize RTC / KRTC components) would limit the scalability and integration into the existing real time control system. Similarly Java, which offers a wide variety of native libraries, features, and GUI capabilities, would leave us unable to fully utilize the RTC framework. The remaining language choice was C++.

The current V2 FDL Sequencer is written in C++; this provides us with the advantage of being able to simply duplicate the calculation routines to the ASTRA Sequencer, rather than having to build a separate stand-alone library as Python or Java would require. Implementing with C++ would also allow us to utilize the entire suite of RTC / KRTC components, including dynamic configuration and project management.

As with other RTC and KRTC components, we will be using CORBA as the primary means of communication and commanding for the FDL sequencers. The interface of the FDL Sequencer is defined as an IDL class definition, which will be compiled and implemented as a CORBA servant. The CORBA servant itself will be designed as a finite state machine using the C++ State-Map Compiler (SMC) to create the necessary state-transition framework.

The following packages / tools should be used for building and development;

- ACE TAO 1.4.1 for Solaris
- GNU C++ 3.2 or newer for Solaris
- GNU Make 3.79 or newer
- SMC 5.0 or newer
- Java 1.5 or newer (required by SMC)

3.0 Sequencer Interface

The ASTRA sequencers will make use of CORBA and RTC Telemetry to facilitate communication between the sequencers and clients. CORBA will be used primarily as the command interface to the FDL Sequencer. Telemetry will provide sequencer feedback to the Observing Sequencer and the operator GUIs. This feedback will be used to determine the next appropriate step, inform the user of the status of issued commands, and report other information including the current target and state.

The CORBA interface to the FDL Sequencer is outline in the following section. Telemetry Channels are discussed in section 3.2.

3.1 IDL Interface

The FDL Sequencer CORBA servant will provide the main interface for user control. As a CORBA servant, operators will have the ability to start and run the FDL Sequencer independently of the other ASTRA components, and can even execute the sequencer on other hosts to distribute network, memory, and processing resources as required. The following details the IDL interface.

```
// Defines the base sequencer class.
#include "SequencerSubsystem.idl"

// Defines the Target Manager interface and related structures.
#include "TargetManager.idl"

// FDL Target Generator is part of ISEQ (IF Sequencers)
module iseq {

    interface fdlSubsystemInterface : SequencerSubsystemInterface
    {

      enum ModeType {
          SYMMETRIC,        // FDL0 delay is negative FDL1 delay
          FDL1_CENTERED,    // FDL1 fixed, only FDL0 tracking
          FDL1_OPTIMIZED,   //
          TEST              // generate ramp targets for both delay lines
      };

      typedef sequence<ModeType> ModeTypeSeq;

      exception InvalidIndex {};

      // Stop the FDL sequencer and put it into standby mode.
      void Standby();

      void Acquire(in TargetManager::TargetInfo Coordinates);

      // Puts the delayline into track mode.
      void Track(in long DelayLine) raises(InvalidIndex);

      // Idles the delayline.
      void Idle(in long DelayLine) raises(InvalidIndex);

      // Zero the FDLs
      void Zero();
```

```
      // Set the operating mode
      void SetMode(in ModeType Mode);

      // Set the LDL optical path difference.
      void SetLDLOPD(in double opd);

      // Enable / disable LDLOPD calculation
      void SetLDLOPDEnable(in boolean enable);

      void SetManualOffset(in double Offset, in double DelayLineIndex)
               raises(InvalidIndex);

      void SetOffset(in double Offset);

      // Shutdown the FDL sequencer.
      void ShutdownSequencer();

   };
};
```

The FDL Sequencer inherits its base Sequencer class definition from the ISEQ Sequencer Subsystem Interface. This component provides the common sequencer operations *Halt*, *Init*, and *ReInit*. In addition to the inherited sequencer commands the FDL Sequencer implements a *Standby* and *ShutdownSequencer* method to idle and shutdown the sequencer object. Other than the standard state control operations, the FDL Sequencer interface defines the command structure for the Target Generation and delay line components. Target Generation is initiated through a call to the *Acquire* method, which takes a set of target coordinates for its argument. The *SetMode*, *SetLDLOPD*, *SetLDLOPDEnable*, *SetManualOffset*, and *SetOffset* methods allow clients to explicitly configure the target generator at run-time. The interface also provides an *Idle* and *Track* method which act as simple pass through operations to the specified delay line gizmo.

The non-trivial sequencer methods are described below.

3.1.1 Standby

A call to the *Standby* method will put the sequencer in to the *STANDBY* state. In the *STANDBY* state, target generation will be suspended and the sequencer will simply be waiting for a user command.

3.1.2 Acquire

The *Acquire* method is designed to put the sequencer in to the *ACQUIRE* state, and begin target generation. The ACQUIRE state is responsible for starting the periodic generation process, which will run as a separate thread of execution. Each FDL sequencer will get a different target determined by the higher level ASTRA Observing Sequencer. The FDL Sequencer will periodically generate Delay Line positions based on the provided target.

3.1.3 Zero

The *Zero* method is designed to put the sequencer into the *ZERO* state, where it will set the internal state to default values, and set the delay line back to the home position. This state can also be used to reset the position of the FDL, or perform other cleanup / baseline operations.

3.1.4 ShutdownSequencer

This method will cause the FDL Sequencer to stop all of its target generation, cleanup the class state and terminate the state processing thread, effectively shutting down the FDL sequencer.

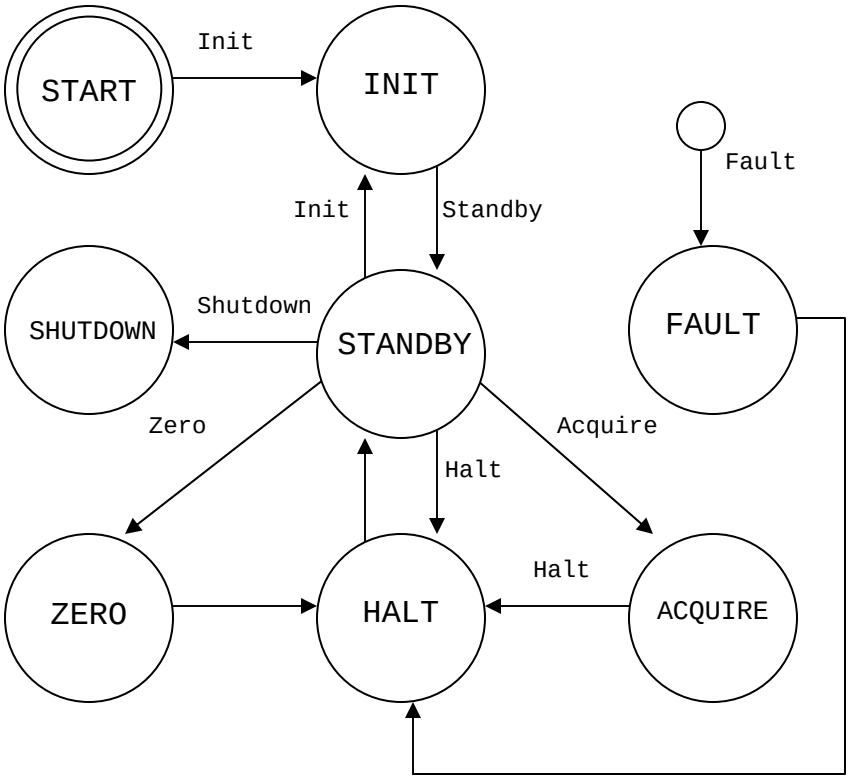
3.2 Telemetry

RTC Telemetry channels will be used to publish observing information, command results, and general sequencer state information. Any RTC Telemetry Consumer client can subscribe to this information to monitor the progress of a sequence and the health of the various components.

The following telemetry channels will be created for the FDL Sequencer,

- *State* – (string) The current state of the FDL Sequencer.
- *Status* – (string) The overall status of the FDL Sequencer. This can be one of
    - `CREATED` – The Sequencer has been created, but not yet initialized.
    - `RUNNING` – The Sequencer is running and able to accept commands.
    - `ERROR` – A command or operation has failed. See the *ErrorMessage* item for more details.
    - `SHUTDOWN` – The sequencer has been shutdown.
- `ErrorMessage` – (string) This item will publish an error message that can be used to diagnose the last fault.
- `HeartBeat` – Regularly publishes an incrementing value to identify that the Sequencer is still alive.
- `Comment` – (string) Baseline comment. This item is typically set in the database and provides a brief description of the observing configuration.
- `TestRateSupplier` – (double) Publishes the test rate used when the FDL Sequencer is in *ModeType::TEST* mode.
- `StarId` – (string) The name of the current target.
- `Offset` – (double) Publishes the value set in the call to *SetOffset*.
- `ManualOffset0Tel` – (double) Publishes the value set in the call to *SetManualOffset* for FDL 0.
- `ManualOffset1Tel` – (double) Publishes the value set in the call to *SetManualOffset* for FDL 1.
- `LDLKeywordEnableTel` – (int) Publishes the value set through the call to *SetLDLOPDEnable*.
- `LDLOPD` – (double) The current LDL optical path difference.
- `Mode` – (fdlSubsystemInterface::ModeType) The current operating mode.
- `Coord` – (TargetManager::TargetInfo) Coordinate information for current target.
- `Baseline` – (SeqTelemetryModule::Baseline) Information on the current baseline configuration.
- `Target0` – (KeckDelayLineModule::Target) Delay line 0 target information.
- `Target1` – (KeckDelayLineModule::Target) Delay line 1 target information.

4.0 Sequencer State Machine



The FDL Sequencer state machine is a distinct software object and is created through the SMC state-map compiler. This object doesn't actually implement the code that will be executed by the FDL Sequencer. Instead it implements a transition-callback mapping that will be used to execute specific methods implemented by a separate class. In the case of the FDL Sequencer this class will be the CORBA servant. (In essence the servant will be handling two duties; receive CORBA requests to initiate state-machine transitions, as well as implement the code that will be executed by that transition).

The following sections detail each of the FDL Sequencer states, and what operations and duties each will perform.

4.1 START

When the FDL Sequencer CORBA servant is started it will create and initialize the state machine and local members, as well as bind itself to the Naming Service. Connections will be made to the associated Delay Line gizmos based on information from the Configuration Server. Once complete the sequencer will be ready for initialization.

4.2 INIT

The *INIT* state is responsible for initializing sequencer state variables and publishing the entirety of the sequencer telemetry. The FDL Sequencer must be initialized before it can be used to generate targets.

4.3 STANDBY

The *STANDBY* state begins and terminates all state transitions, and while in *STANDBY* no target generation is preformed. From this state any of the other commandable states can be accessed

4.4 ZERO

The *ZERO* state is designed to zero out the target position, rate and the timestamps used to perform target generation calculations.

4.5 ACQUIRE

The *ACQUIRE* state performs all of the FDL target position generation for the sequencer. When a user calls the *Acquire* method on the CORBA servant they will provide the target information for the selected stellar object. This information will be saved in a class member accessible by the state machine. A call to *Acquire* will cause the servant to issue an *ACQUIRE* state transition, putting the processing thread into the *ACQUIRE* state. On entry of this state an RTC periodic task will be started. This task will be responsible for performing the actual position calculations, and offloading them to the FDLs.

An FDL target position is based on a specific projected time in the future. This times is determined by the formula,

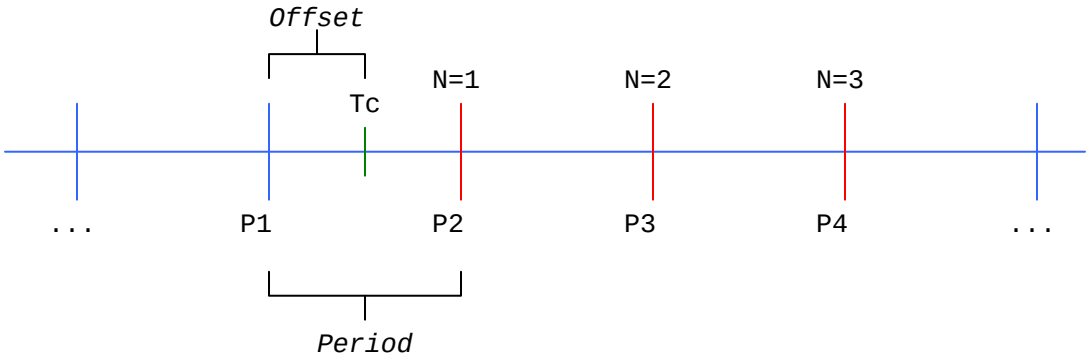
$$(Tc - (Tc\%P)) + N * P = Tt$$

Where *Tc* is the current time (seconds), *P* is the period (seconds), *N* is the offset (period counts), and *Tt* is the target time (seconds). Assuming each FDL Sequencer uses the same period and offset, all target positions will be produced for the same target time; effectively synchronizing each Delay Line.

To ensure that the calculations are performed at a specific predictable rate we will implement an `RTC::PeriodicTask` to execute the calculation routine. Periodic Tasks lend themselves nicely to the FDL Target Generation process because they provide very accurate periodic execution and also provide an offset mechanism, which will allow the operator to control when during a period the calculation is performed. Since all periodic tasks started under the same CPU Manager share a single Scheduler, we can be confident that each FDL sequencer will be performing target generations in synch. The following formula is used by the Scheduler to determine if the periodic task is ready to execute.

$$(Tc\%P) + Offset \geq Ts$$

Where *Tc* is the current time (nanoseconds), *P* is the period (nanoseconds), *Offset* is a delay time before the task begins execution (nanoseconds), and *Ts* is the scheduled time. The RTC Scheduler is responsible for performing this calculation and starting the task execution. The following diagram describes the target generation process on a timeline.



The Scheduler fires off the periodic task at the start of a period (*P1*) plus the specified delay (*Offset*). The periodic task begins the target generation process at time *Tc*. The calculation routine will compute a target position for a timestamp based on the period offset, *N*. If this offset is zero, it will calculate a position based on the timestamp for the start of the next period (*P2*). If the offset is equal to one, it will calculate the position for the timestamp that is equal to the start of the following period (*P3*).

When the *ACQUIRE* state is left through a transition to *IDLE* the periodic task is disabled, and will only be re-enabled on entry to *ACQUIRE*.

4.6 HALT

The *HALT* state is responsible for putting the sequencer into a safe state before automatically transitioning to the *STANDBY* state. As the main function of the FDL Sequencer is to generate targets, the *HALT* state takes the action of disabling the periodic task if it is running.

4.7 FAULT

The *FAULT* state is responsible for catching operational errors that occur during sequencer execution. Any state can be transitioned to the *FAULT* state, but it can not be entered explicitly by user command. The *FAULT* state will report the cause of the error through the error telemetry item, and then automatically transition to *HALT* to ready the sequencer.

4.8 SHUTDOWN

The *SHUTDOWN* state should be called when observing for the night is complete, and the sequencer is ready to be shut down. This state will halt FDL target generation, delete the periodic task, destroy the state machine, and cleanup the CORBA servant. After the *SHUTDOWN* state is executed the

FDL sequencer will no longer be able to respond to state transitions until the CPU Manager that hosts the sequencer is restarted.

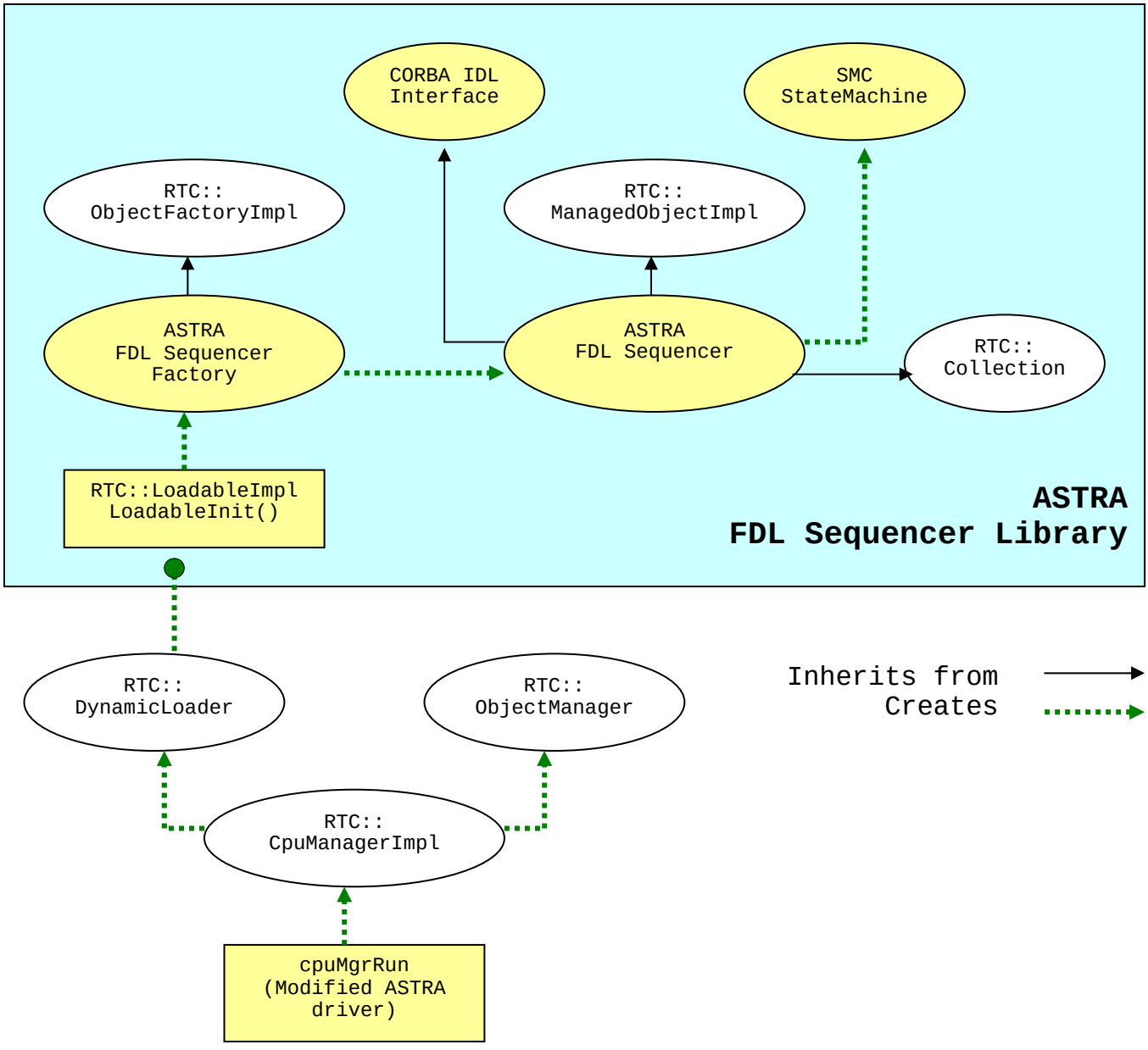
5.0 Developing with RTC

The following sections detail the design issues, patterns, and requirements involved when developing with the RTC framework. As the FDL Sequencer will rely on RTC for task management, configuration, and initial creation we must build the design around the basic components and usage requirements of RTC. The majority of this design will be based on the rules for developing CPU Manager compatible code. If we can succeed in designing the FDL Sequencer to run through a CPU Manager we will have access to the majority of the RTC features.

For more information on developing with the CPU Manager see the appendix for links to handover and reference material.

5.1 Object Diagram

The following diagram details the object inheritance and component interaction of the ASTRA FDL Sequencer based on the RTC framework. Items in yellow represent components that will need to be implemented specifically for the project. Items in white represent the existing RTC infrastructure that will not require changes.



(Figure 1)

5.2 FT Sequencer Components

The FDL Sequencer implements the sequencer state-machine, CORBA interface, and the low-level collection of the RTC component hierarchy.

5.2.1 CORBA Servant

The CORBA interface provides users with remote invocation capabilities allowing clients to command and control the sequencer from GUIs or simple applications. An object reference for the CORBA servant will be bound to a specific name context in the Naming Service. This context is created during the Sequencer creation, and will provide easy lookup and access for remote users. The FDL Sequencer will implement the stubbed C++ CORBA interface, as is done with the existing CORBA components used throughout RTC and KRTC.

5.2.2 State Machine

The state machine logic is produced as part of the SMC API, which allows a developer to define states, transitions, and actions for each desired event. The state mappings will be compiled into a set of C++ class utilities that will be utilized by the FDL Sequencer to provide state transition capabilities. The CORBA servant will have an instance of the state machine definition as a class member, and will also implement the associated state-actions, which the state machine will execute.

5.2.3 Managed Objects

As the sequencer will need to make use of the RTC framework it will have to inherit from some of the low level RTC components. To allow for the dynamic creation of sequencer instances by the CPU Manager, it will need to inherit the `RTC::ManagedObjectImpl` definition. This class will provide the inheriting class with the ability to be created by an object factory (the ASTRA FDL Sequencer Factory), as well as enforcing a constructor format that assigns a name to the object instance (`RTC::ManagedObject::Id`).

5.2.4 Collections

The `RTC::Configuration::Collection` object provides classes with a built in naming and object hierarchy, as well access to the dynamic reconfigurable `RTC::Configuration::Entry` item. The `Collection` object also provides a means by which to notify the `Entry's` parent (the FDL Sequencer) that a configuration has occurred through the *PostVisit* method.

This is a critically important feature for the ASTRA sequencers because connections to the Naming Service, gizmos, and keywords can only be performed after the configuration entry items have been set with their actual configuration values. The FDL Sequencer will override the *PostVisit* method with an initialization routine that will make these connections. The initialization routine will be responsible for binding the Sequencer instance to the Naming Service and connecting to the Delay Line gizmo.

5.3 FDL Sequencer Factory

In the RTC world, object instances are created by the CPU Manager through the use of object factories. For the ASTRA FDL Sequencer this will be done through the FDL Sequencer Factory.

One factory exists for each type of object within an individual RTC module (a module is a loadable library file). Factories implement a simple interface defined by the `RTC::ObjectFactoryImpl` class, which in turn inherits the `RTC::LoadableImpl` definition. These classes provide the CPU Manager with the ability to create and initialize the FDL Sequencer factory from a module file, and it gives the factory the ability to create instances of the desired object on demand. For the purposes of the ASTRA FDL Sequencer factory we will only need to implement the inherited *Create* method to actually instantiate and return an instance of an FDL Sequencer; the other methods are satisfactorily handled by the inherited RTC implementations.

5.4 Loadable FDL Module

The RTC CPU Manager is designed to load precompiled library files, and through the use of a bootstrapping function, incarnate the objects built within. The boot strapping function has the form `RTC::LoadableImpl* LoadableInit()` and is defined as a ‘C’ function. For the FDL Sequencer, this function will create an instance of the sequencer factory and return a pointer to the instance. The actual process of loading the library and creating and initializing the factory is done through the RTC Dynamic Loader service. For more information on the Dynamic Loader or other CPU Manager services see the CPU Manager handover documentation.

In order to allow the CPU Manager to load and create an instance of the FDL Sequencer the CORBA servant, state-machine, FDL Sequencer factory, and bootstrapping function will need to be built into a dynamic library. This is done with the following G++ compiler commands:

```
g++ -fPIC <include_paths> -c <source_file> -o <object_file>
g++ -shared <object_files> -o <lib_name> -L <library_files : ACE>
```

6.0 Configuration

The RTC Configuration Server provides the FDL Sequencer with configuration information through the *Configuration::Entry* implementation. All of the configurable parameters and values will be maintained by the Configuration Database, and will be applied to the Sequencer during the CPU Manager startup process. This section outlines the configurable items of the FDL Sequencer.

6.1 Configuration Layout

The Configuration Database defines object information as a nested hierarchy of items. Each sub-level of a hierarchy represents a child component with its own configurable parameters. Object definitions can also inherit other objects, effectively providing OO configuration support. The FDL Sequencer is based on the base configuration class *SequencerSubsystem*, and will make use the

complex types *PeriodicTask* and *tTelemetryItem*. The FDL Sequencer inherits the following items from the *SequencerSusbystem*:

- *DebugLevel* - (long) Sets the debug print level used to control the level of print information during runtime.
- *ErrorMessage* – (TelemetryItem) Provides error message publishing.
- *SequencerName* – The dot-delimited name used to register the sequencer with the CORBA Naming Service.
- *Simulate* – (long) Defines whether the sequencer should be run in simulate mode (interpreted as boolean).
- *State* – (TelemetryItem) The current state of the sequencer.
- *Status* – (TelemetryItem) The current status of the sequencer.

On top of the *SequencerSubsystem* base class, the *FDLSequencer* class defines the following configuration items:

- *AZDlGain0* – (double)
- *AZDlGain1* – (double)
- *AZDlOffset0* – (double)
- *AZDlOffset1* – (double)
- *AZDlPhaseOffset0* – (double)
- *AZDlPhaseOffset1* – (double)
- *AZDlRateSign0* – (long)
- *AZDlRateSign1* – (long)
- *AZKeyword* – (string) The KTL keyword for the telescope azimuth.
- *AZOffsetEnable0* – (long)
- *AZOffsetEnable1* – (long)
- *AZRateUpdate* – (long)
- *AcquisitionTask* – (PeriodicTaskImpl) The RTC periodic task used to generate FDL targets.
- *Baseline* – (TelemetryItem) Publishes the baseline information.
- *BaselineComment* – (string) A comment describing the baseline configuration.
- *BaselineE* – (double)
- *BaselineN* – (double)
- *BaselineOffset* – (double)
- *BaselineUp* – (double)
- *Comment* – (TelemetryItem) Publishes the baseline comment.
- *Coord* – (TelemetryItem) Publishes the target information.
- *DCSService* – (string) The KTL service name for the DCS service.
- *FDLName0* – (string) The dot-delimited name for FDL ‘0’ found in the CORBA Naming Service.
- *FDLName1* – (string) The dot-delimited name for FDL ‘1’ found in the CORBA Naming Service.
- *LDL1Keyword* – (string) The KTL keyword for LDL 1.
- *LDL2Keyword* – (string) The KTL keyword for LDL 2.
- *LDLKeywordEnable* – (long) Defines whether LDL OPD calculations should be performed (interpreted as a boolean).
- *LDLKeywordEnableTel* – (TelemetryItem) Publishes whether the LDL OPD calculations are enabled.

- *LDLOPD* – (TelemetryItem) Publishes the current LDL OPD.
- *ManualOffset0* – (double) Sets the manual offset for FDL ‘0’
- *ManaualOffset0Tel* – (TelemetryItem) Publishes the manual offset for FDL ‘0’.
- *ManualOffset1* – (double) Sets the manual offset for FDL ‘1’
- *ManaualOffset1Tel* – (TelemetryItem) Publishes the manual offset for FDL ‘1’.
- *Mode* – (TelemetryItem) Publishes the current operating mode.
- *Offset* – (TelemetryItem)
- *OffsetForeground* – (double)
- *OffsetMidpoint* – (double)
- *PeriodicCountOffset* – (double) Used by the target generation task to determine the period displacement when calculating FDL target for timestamps in the future.
- *StarId* – (TelemetryItem) Publishes the current target star name or ID.
- *TOService* – (string) The KTL service name for the TO service.
- *Target0* – (TelemetryItem) Publishes Delay Line target information for FDL ‘0’.
- *Target1* – (TelemetryItem) Publishes Delay Line target information for FDL ‘1’
- *TestRate* – (double) Sets the test rate for operating in *ModeType::TEST*.
- *TestRateSupplier* – (TelemetryItem) Publishes the test rate.

7.0 A Typical Sequence

The following outlines a typical FDL operating sequence as controlled by the higher level Observing Sequencer.

7.1 Startup

The FDL Sequencer is created and initialized through a modified version of the Generic CPU Manager startup application, *cpuRun*, as outlined in section 5.0. The CPU Manager will be provided the name of the root configuration item that defines all of the libraries and object that will be needed by the FDL Sequencer. This process can be initiated by a general ASTRA system startup script, or it can be handled by the ASTRA Observing GUI.

Once the FDL Sequencer has been created it will immediately enter its *START* state where it will register with the CORBA Naming Service, connect to the Telemetry Server, obtain FDL CORBA references, and initialize its remaining members before transitioning to the *IDLE* state. It will wait in this state until the Observing Sequencer has entered the target acquisition phase.

7.2 Acquire Targets

When the Observing Sequencer is ready to initiate the target generation process it will call the FDL Sequencer's *Acquire* method with the appropriate target information. It is the responsibility of the Observing Sequencer to determine the appropriate coordinates; allowing the FDL Sequencer to focus on converting those coordinates into Delay Line positions.

When the *Acquire* method is called the target information will be saved to a member of the CORBA servant, and the *Acquire* transition is executed for the internal state machine. The transition will execute a corresponding class method (e.g. *StateAcquire*) that will be responsible for starting the target generation periodic task. At this point the main CORBA / State Machine thread will be idle until another IDL call to transition out of the *ACQUIRE* state is executed by the Observing Sequencer.

The periodic task however will continue to work in the background for the duration of the *ACQUIRE* state. The RTC Scheduler will execute the task code at the rate defined in the configuration database - at most once a second. During each execution, new Delay Line positions will be calculated based on the target coordinates and the projected time, and sent to the corresponding Delay Lines. These values will be combined with other offsets received by the Delay Liens from other sources, and will be applied to the track position at the projected time.

7.3 End Target Generation

The target generation process will terminate when the FDL Sequencer receives a call to transition to the *IDLE* state. As with the *Acquire* operation, a call to *Idle* will cause the state machine to execute a corresponding idle method (e.g. *StateIdle*). This method will be responsible for halting the periodic task and for removing it from the Scheduler's active task list. The state machine will then enter the *IDLE* state to wait for the next command.

7.4 Shutdown

When the night's observing is complete the FDL Sequencer's *Shutdown* method will be called, causing the Sequencer to execute the shutdown operation. This will involve releasing CORBA and Telemetry references, deleting the periodic task, and cleaning up the CORBA Servant and State Machine members. The FDL Sequencer will remain in a disabled state until the CPU Manager is finally shutdown, releasing memory and system resources.

8.0 Work Estimate

We estimate approximately 70 hours to implement and 16 hours test the ASTRA FDL Sequencer upgrade.

Additional Resources

ASTRA Homepage

RTC CPU Manager Handover Twiki

ASTRA CPU Manager Upgrade Plan (MS Word Document)

Target Manager Documentation (MS Word Document)