LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)
Telescope & Site

Control Software Architecture

Tiago Ribeiro, William O'Mullane, Tim Axelrod, Dave Mills

LSE-150

Latest Revision: 2019-02-03

Draft Revision NOT YET Approved – This LSST document has been approved as a Content-Controlled Document. Its contents are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. If this document is changed or superseded, the new document will retain the Handle designation shown above. The control is on the most recent digital document with this Handle in the LSST digital archive and not printed versions. – Draft Revision NOT YET Approved

Abstract

TSS Architecture and approach.

Change Record

Version	Date	Description	Owner name
1	2012-12-14	V1	German Schumacher
	2019-01-20	Unreleased.	William O'Mullane, Tim Axelrod
	2019-02-01	Unreleased.	Tiago Ribeiro

Draft

Contents

Draft

Control Software Architecture

1 Introduction

The LSST Control Software contains the overall control aspects of the survey and the telescope including the computers, network, communication and software infrastructure. It contains all work required to design, code, test and integrate, in the lab and in the field, the high level coordination software.

2 System Architecture

The LSST control system is based on a reactive data-driven actor-based architecture that uses a multi cast Data Distribution Service (DDS) messaging protocol middleware. A high level view of this architecture is given in Figure 1, where each box corresponds to a component of the system (not all components are displayed here).

The LSST System Architecture is comprised mainly of;

- The Service Abstraction Layer (SAL[1]) communication middleware. Based on the DDS protocol, it provides interfaces for all the project adopted programming languages (LabView, C++, Java and Python).
- Engineering and Facility Database (EFD).
- SAL-aware reactive components, a.k.a Commandable SAL Components (CSCs).
- LSST Operators Visualization Environment (LOVE).

The SAL middleware is the backbone of the LSST system architecture. It implements three distinct types of messages; Commands, Events and Telemetry, with distinct purposes. Commands are sent to a specific component, which must acknowledge its receipt and perform some action. In general, the receiving component will be the only entity listening for the commands it accepts. Events and Telemetry are messages broadcast by components to the middleware and are available to any entity on the system to receive. The distinction between

[1] https://docushare.lsstcorp.org/docushare/dsweb/Get/Document-21527/

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

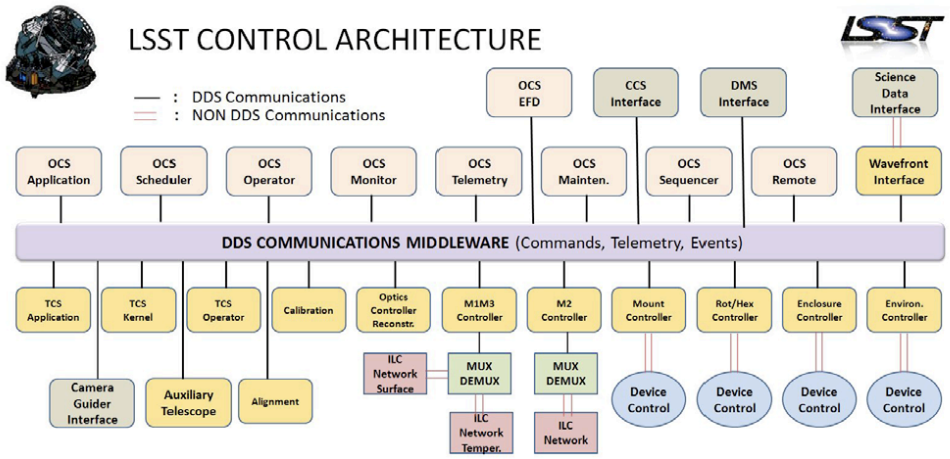

FIGURE 1: High Level Architecture Diagram

Events and Telemetry is that the former receives a higher priority by the message passing systems.

The EFD is responsible for capturing all SAL messages broadcasts to the middleware (including Commands, Events and Telemetry) and storing that information into a database.

CSCs are the main actors of the LSST system architecture. They are responsible for managing the incoming traffic of data and take appropriate actions, controlling hardware (e.g. M1M3, M2, Mount Controller, etc in Fig. 1), software (e.g. Optics Controller Reconstructor, DMCS Interface, etc in Fig. 1) or even other CSCs (e.g. Script Queue, TCS, ATCS, OCS, etc in Fig. 1).

LOVE is responsible for capturing SAL messages and displaying them in a useful way for general users, providing some basic interface to query and analyze data from the EFD and an interface to issue some pre-defined commands to a set of components.

One of the fundamental parts of SAL is to provide a low-level API for publishing and subscribing to the middleware. These APIs are generated for each component independently, based on pre-defined interfaces. On top of those low level APIs, developers have access two higher level set of frameworks; Python SalObj[2] library and the LabView component template. No higher level framework is supported for implementation in Java or C++.

Overall, the system architecture can be divided into three main namespaces; Observatory,

[2] https://github.com/lsst-ts/ts_salobj

Main Telescope (MT) and Auxiliary Telescope (AT). The Observatory is the highest level and encapsulates both the Main Telescope, Auxiliary Telescope and global components such as the weather station, DIMM, etc. The complete set of components that belong to each of these namespaces can be seen in Figs. **??**, 2 and 3.

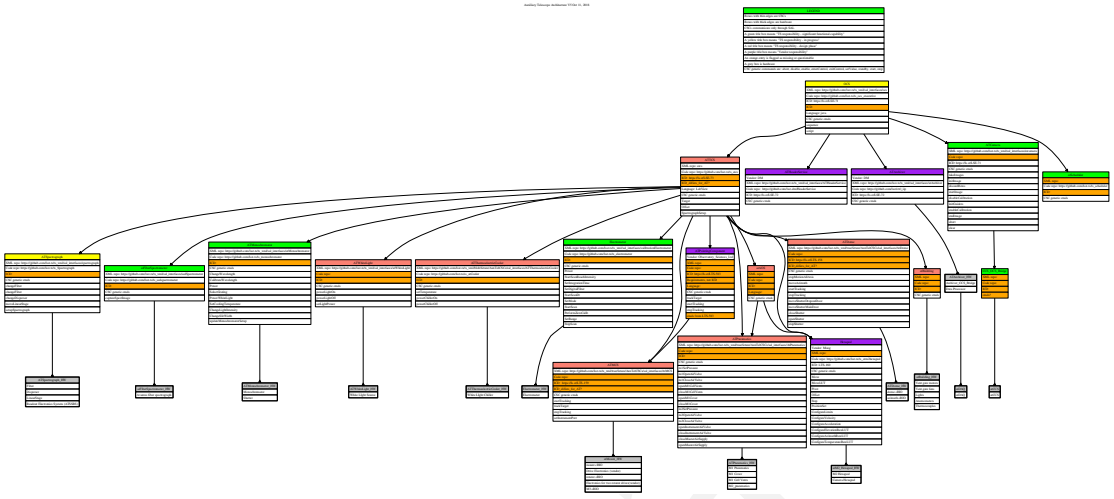

FIGURE 2: Complete set of AT CSCs

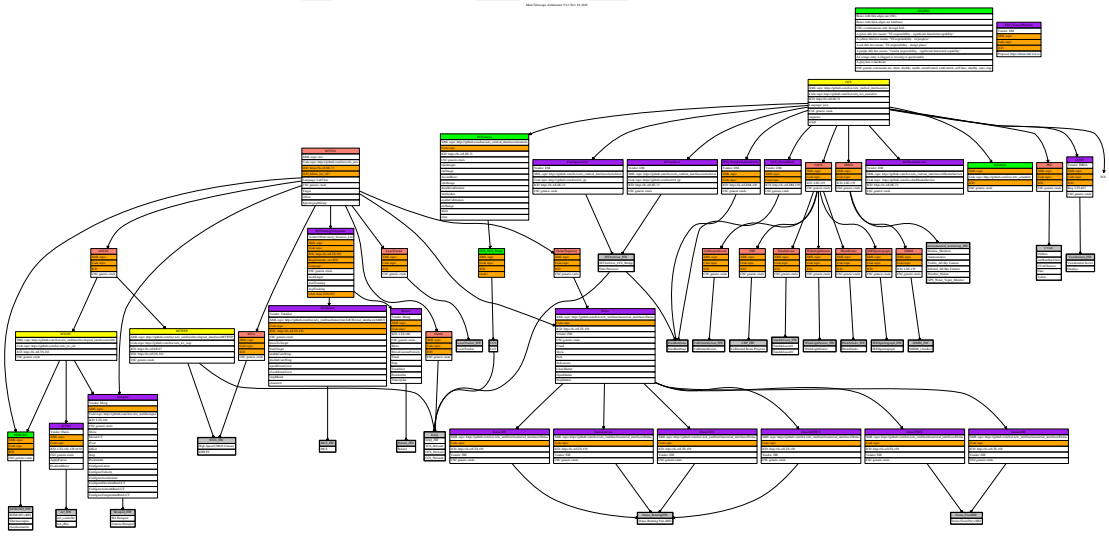

FIGURE 3: Complete set of MT CSCs

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

2.1 SalObj - Python and scripting

SalObj is a Python library provides a pythonic and object-oriented interface to create CSCs and Scripts that can be executed by the script queue component (see Sect. 2.3.1). The library defines two sets of base classes that are mirror to each other, Remote and Controller. A Remote will send commands to and receive telemetry and events from a specific component whereas a Controller will receive commands and publish telemetry and events. In this framework, a CSC is a specialized Controller that is configure to perform some basic actions by default. A high level diagram is provided in Figure 4.

Internally, SalObj uses the python library `asyncio`[3] to handle the inherently asynchronous nature of the SAL messaging system.

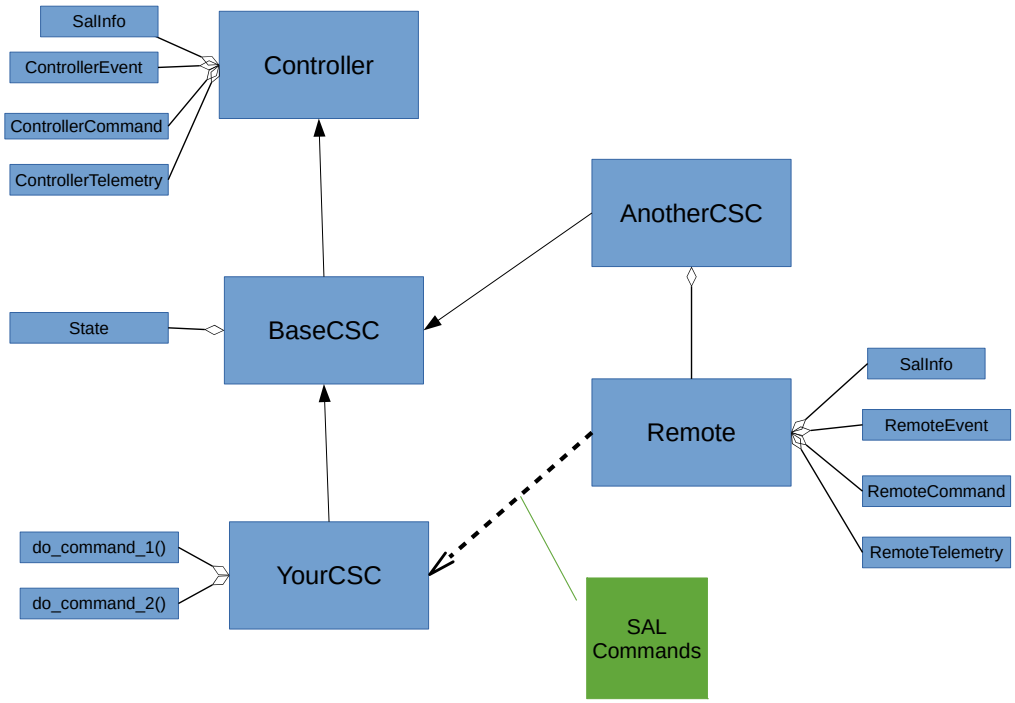

FIGURE 4: SalObj python scheme for CSCs

[3] `https://docs.python.org/3/library/asyncio.html`

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

2.2 Hardware interface components

Probably the most critical or sensitive components of the LSST system architecture are those that directly control hardware. Some of these components are going to be delivered directly by external vendors, such as those that will control the main telescope mount (MTMount) and the main telescope secondary mirror (MTM2). There are also those that are developed in house, e.g. the main telescope M1M3 (MTM1M3).

In some special cases it is highly desirable that the control software and hardware are part of an integrated system. For those systems, the components are developed either using the LabView component template, which is part of the LSST infrastructure or in C++ developed using the low level SAL API.

In most other cases, the hardware comes with a control software that can be easily interfaced by using standard protocols (such as TCP/IP or serial ports), and there is no special need for the component software to reside close to the low level hardware controller. In those cases, the components are written in Python using the SalObj library which is also part of the LSST infrastructure. By writing these components in Python we allow a high level of flexibility and maintainability of the software components and considerably decrease the development cycle.

2.3 Pure software components

In the LSST System Architecture there are a number of components that, even tough they do not control hardware directly, they dictate what hardware components are supposed to do. Some of these components are responsible for heavy computational routines, such as the Optical Feedback Control (MTOFC), which is responsible for applying corrections to both M1M3, M2 and hexapod components for the main telescope or even the Scheduler, which is responsible for processing the an entire set of observatory telemetry information and history of observations to compute an observing queue.

These pure software components are mostly written in Python using SalObj library. There are three special cases of these components that form the basis of the LSST System Architecture; the Script Queue (Sect. 2.3.1), Control Systems (Sect. 2.3.2) and the Watcher (Sect. 2.3.3). Together, they provide the tools needed for integration, commissioning and operation of the observatory.

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

2.3.1 The ScriptQueue component

There are a number of different ways users can interact with components in the LSST system. For instance, one could easily use the SAL generated API in any of the supported languages to send commands directly to a single or multiple components. It is also possible to use SalObj Remotes to write Python scripts that would command different components to accomplish a specified task. Not to mention that LOVE itself provides a customizable interface for users to interact with components.

During commissioning and operations the LSST system will require a high degree of coordination between different crews (different daytime and nighttime shifts, for instance), not to mention the increasing number of available components and level of complexity as the system ramps up. In order to manager those issues, the LSST control system contains a specialized script queuing component, a.k.a. the ScriptQueue[4].

The ScriptQueue defines an interface for developing scripts in general, and provides a basic class that can be used to develop Python scripts. As Python programs, these scripts have access to all Python functionality, both from the native Python 3 language and through imported modules (including `asyncio` to manage concurrent activities or libraries from the DM stack). In particular, a Script has access to all the system components using SalObj Remotes (Section 2.1) . Although Python is the only language officially supported, scripts can be written in any SAL-supported language. As long as they follow the interface defined by the ScriptQueue component, it should be possible to execute them.

2.3.2 Control Systems

In such an environment it is not immediately clear that a traditional hierarchical design is necessary or desirable. A completely flat architecture initially seems completely workable and certainly sufficient during AIT and early commissioning. For example, consider a Script which commands and sequences the telescope subsystems to move to the next field to be observed, take an exposure, and read out that exposure. The Script can directly control each of those subsystems and maintain control of the sequencing using `asyncio`. Furthermore, the complexity of the Script can be managed through normal modular programming techniques, in which subsystem functionality is implemented through Python objects imported in modules.

Though we could have a flat system based on ScriptQueue (Section 2.3.1) there are two com-

[4] `https://github.com/lsst-ts/ts_scriptqueue`

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

pelling reason to retain at least a top level OCS which has overall responsibility for all subsystems. The first reason is rooted in the limited lifetime of each Script, which has an execution thread that begins and ends over a duration short compared to the up time of the telescope system. When a Script is instantiated, it has no immediate knowledge of the state of the telescope system as a whole, or the state of individual subsystems. It needs such knowledge, because many actions, e.g. moving the telescope, require the telescope subsystems to be in particular states. The Script can, of course, assemble the required knowledge, by using SAL to obtain the state information of all relevant subsystems, but doing so imposes unnecessary startup overheads for Scripts. It is far more efficient, not to say reliable, for a single subsystem to be continuously responsible for maintaining knowledge of the overall state of the observatory (observatory states are discussed in more detail below).

The second, related, reason is that the operator needs:

1. to maintain continuous knowledge of the state of the observatory independent of Script execution, and
2. to be able to command the observatory to change its overall state.

An excellent example of the latter requirement is from the LOVE requirements document, LTS-807:

> **Requirement ID:** LOVE-REQ-0078
> **Requirement Title:** Emergency Close
> **Specification:** The LOVE **shall** provide a single control command labeled "Emergency Close" of the telescopes.

Certainly one could imagine creating a Script to execute Emergency Close, but to use it would require that any currently running Script be aborted, and the script to be placed on top of the queue which may create a dangerous overhead.

By adding high level Control Systems that are responsible for monitoring the heath and state of a group of components, it is possible to increase overall system responsiveness, simplify script development cycle and so on. This is accomplished by the use of a Generic Control System (GCS) component.

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

A GCS defines a namespace which contains a group of component to be supervised. The list of components that are part of a Control System group is a configurable parameter. The Control System component still operates normally, taking the appropriate actions, if one or more of the components in its group is missing, unresponsive or in a fault state. The Control System manages and oversee the state of each component in its group.

For each component, there are a set of basic actions that a Control System is capable of performing and it is possible to expand and customize these actions for a specific component (or group of components). An example of a default action is to bring components up to an enable state, configuring it in the process. The decision to bring a component up can be either following a request to use the component by an external agent or client (e.g. a script), or given some external constraint (e.g. the night is about to start so the Control System enables every component, or a subset of components, in its group).

The ability to define customized actions is an important feature of Control System. It enables one to port actions that are developed inside scripts to control a group of components into a high level command that is performed by the Control System. Since the Control System monitors the state and health of the components inside its namespace and has readily access to the commands they accept, it is possible to execute these high level commands at a faster rate than that achieved from a script alone.

Overall the combination of the ScriptQueue and GCS gives the LSST system a high level of flexibility, speed and reliability.

2.3.3 The Watcher

The Watcher is a component that monitors the other SAL components and output alarms in a standard way that LOVE can present to operators. The Watcher is designed in such a way that alarm rules are easy to write and easy to understand. The rules are likely to evolve rapidly during commissioning and slowly after that.

Examples of alarms published by the Watcher are;

- Dangerous weather, such as rain or high humidity.
- A SAL component is unavailable: not enabled or heartbeat is missing.
- Actuator malfunction, such as axis motors out of closed loop, filter changer stuck, an

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

actuator hits a limit.

- CCD temperatures or pressures out of range.

A typical life cycle of an alarm:

1. Azimuth goes out of range so the controller halts motion. The Watcher reports this as an alarm with severity=serious. LOVE displays it.
2. An operator acknowledges the alarm to the Watcher, but the axis is still out of range. The Watcher outputs a new version of the alarm that includes the information that the alarm has been acknowledged. LOVE displays the alert in a way that looks less urgent (e.g. is grayed out). The alarm has been acknowledged but the condition is still current.
3. An operator fixes the problem and the controller reports this. The Watcher reports the alarm one last time with severity ”OK”. LOVE removes the alarm from the display.

A typical life cycle of a transient alarm:

1. The azimuth drive temporarily draws too much current; the component reports this but manages to keep the axis moving (presumably with temporarily degraded accuracy). The Watcher reports this as an alarm with severity=serious. LOVE displays it.
2. The drive current is within normal range again before an operator has time to acknowledge the alarm. The Watcher outputs a new version of the alarm that says the condition is now OK but the alarm has not yet been acknowledged (a ”stale alarm”). LOVE still displays the alarm, but in different way to indicate that the problem is gone.
3. An operator acknowledges the alarm to the Watcher. The Watcher outputs a new version of the alarm with severity ”OK” and acknowledged=True. LOVE removes the alarm from the display.

2.4 Configuration Management

During commissioning and operations, LSST will have a large number of running software components under the purview of DM, Camera, and TSS. In general, the behavior of each of these components is modifiable through configuration information which is read in during

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

startup of the component, or possibly changed while the component is running. Careful management of this configuration information is crucial to reliable functioning of the Observatory, and to the analysis of its data products.

In this context, git is the solution adopted to store and manage different sets of configurations and different versions of configurations. Git is already a standard in industry as a software management tool and has becoming increasingly used to manage general documents and files as well, not to mention that it is already readily available and broadly adopted by the project. Therefore, each component must establish a **separate** git repository to store its configuration files.

These configuration repositories will be hosted on a configuration server at the summit so that, even if communication with the base or the internet is not available, components still maintain access to their configuration repositories. Different configuration sets (or labels) are stored in separate branches, and tags can be created to specify immutable sets.

Several options for configuration file format, and their associated software tools, have been considered. Each of the available options naturally has its strengths and weaknesses, and none stand out as being particularly useful for all LSST use cases (and/or available for all the project adopted programming languages).

For components written in Python, `pex_config` is the adopted solution. As an overview of `pex_config`, here are a few snippets from the library documentation:

> The `lsst.pex.config` module provides a configuration system for the LSST Science Pipelines.... Configurations are hierarchical trees of parameters used to control the execution of code.... Configurations are stored in instances of a "config" object, which are subclasses of the `Config` class. Different configuration sets are associated with different subclasses of `Config`. For example, in the task framework each task is associated with a specific `Config` subclass.... Configuration objects have `fields` that are discrete settings. These `fields` are attributes on config classes.

In the TSS context the "task" above becomes a "CSC".

`lsst.pex.config.Config` class has methods `save()` and `load()` which persist and restore class instances from files, which just contain Python code. Note that because these files are Python code, it is easy to include documentation within the files.

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED

The validation of a `Config` file is handled by the `__init__()` method of the `Config` subclass which can check, for example, whether parameter values comply with range limits, or whether all required parameters are specified.

In this framework, the configuration schema (or definition) will be developed and stored in the CSC codebase repository (using `pex_config`). As already stated above, a separate repository hosts the actual configuration files which, in the case of `pex_config`, only contains changes to the default set of values.

2.5 Software Deployment Strategy

SAL will be packed as RPMs...

Those components that can be containerized will be packed as Docker containers + Docker hub server at the summit.

What about those that can't? Examples? M1M3? Things that are done in LabView? What about CRIO? What about vendor supplied software?

Puppet can deploy Docker containers. What about the others?

What about the EFD?

Use RPM + Docker + Puppet ...

A References

References

[1] **[LTS-807]**, Serio, A., 2018, *LSST Operations Viszualization Enviroment (LOVE) Requirements* , LTS-807, URL `https://ls.st/LTS-807`

B Acronyms used in this document

Acronym	Description
AIT	Assembly, Integration, and Test

API	Application Programming Interface
AT	Auxiliary Telescope
ATCS	Auxiliary Telescope Control System
C	Specific programming language (also called ANSI-C)
CCD	Charge-Coupled Device
CSC	Controlable SAL Component
DDS	Data Disposition System
DIMM	Differential Image Motion Monitor
DM	Data Management
DMCS	Data Management Control System
EFD	Engineering Facilities Database
GCS	Generic Control System
ID	Identifier
IP	Internet Protocol
LOVE	LSST Operations Visualization Environment
LSE	LSST Systems Engineering (Document Handle)
LSST	Large Synoptic Survey Telescope
LTS	LSS Telescope and Site (Document handle)
M2	Second mirror
MT	Main Telescope
OCS	Observatory Control System
RPM	Revolutions Per Minute
SAL	Services Access Layer
TCP	Transmission Control Protocol
TCS	Telescope Control System
TS	Test Specification
TSS	Telescope and Site Software
k	kilo; SI units prefix for 1E3

DRAFT NOT YET APPROVED – The contents of this document are subject to configuration control and may not be changed, altered, or their provisions waived without prior approval. – DRAFT NOT YET APPROVED