**LSST Service Abstraction Layer (SAL) software SDK**

**Author : Dave Mills**

**Version : 2.2**

**Date : 03/20/2015**

# **Contents**

1. 1 Introduction 2
2. 2 Installation 3
3. 3 Data definition 4
4. 3.1 Telemetry Definition 4
5. 3.2 Command Definition 6
6. 3.3 Log Event Definition 7
7. 4 Using the SDK 8
8. 4.1 Recommended sequence of operations 9
9. 4.1.1 Step 1 – Definition 9
10. 4.1.2 Step 2 – Validation 9
11. 4.1.3 Step 3 – Update structure and documentation 10
12. 4.1.4 Step 4 – Code Generation 10
13. 4.1.5 Step 5 - Verification, Testing and Integration 16
14. 4.2 salgenerator options 17
15. 4.3 SAL examples 18
16. 4.3.1 Example 1- Publishing telemetry 18
17. 4.3.2 Example 2 - Sending a command 19
18. 4.3.3 Example 3 - Receiving commands 22
19. 4.3.4 Example 4 – Generating an event 24
20. 5 Testing 25
    1. 5.1 Environment 25
    2. 5.2 Telemetry 26
    3. 5.3 Commands 26
    4. 5.4 Events 27
    5. 5.5 TCS pointing simulator 29
21. 6 Application programming interfaces 32
22. 6.1 C++ 32
23. 6.2 Java 33
24. 6.3 Python (Boost.python bindings) 34

# **1 Introduction**

This document briefly describes how to use the SAL SDK to generate application

level code to utilize the supported services (Commanding, Telemetry and Logging).

The SAL SDK should install on any modern Linux computer. The current baseline

recommended configuration is 64-bit CentOS 7.0.

The following packages should also be installed prior to working with the SDK

(use either the rpm or yum package managers for CentOS, and apt-get , dpkg,

or synaptic for Debian based systems). Appropriate rpms can be found in the

rpms subdirectory of the unpacked SDK.

- g++

- make

- ncurses-libs

The distribution includes dedicated versions of the following packages

- apache-maven

- boost

- openjdk

- OpenSplice

- python

- tcl/tk

All the services are built upon a framework of OpenSplice DDS. Code may be autogenerated for a variety of compiled and scripting languages, as well as template documentation, and components appropriate for ingest by other software engineering tools.

A comprehensive description of the SAL can be found in doc/LSE74-html, navigate

to the directory with a web browser to view the hyper-linked documentation.

e.g.

firefox file:///opt/doc/LSE74-html/index.html

**2. Installation**

A minimum of 800Mb of disk space is required, and at least 1Gb is recommended

to leave some space for building the test programs.

Unpack the SAL tar archive in a location of choice (/opt is recommended),

e.g. (in a terminal)

cd /opt

tar xzf [location-of-sdk-archive]/salSDK-2.2.1\_x86\_64.tgz

and then add the SDK setup command.

source /opt/setup.env

to your bash login profile.

If you chose to install the SDK in a location other than /opt, then you will need

to edit the first line of the setup.env script to reflect the actual location.

e.g.

LSST\_SDK\_INSTALL=/home/saltester

The most common SDK usage consists of simple steps :

1) Define Telemetry, Command or Log activity (either using the SAL VM, or manually

with an ascii text editor). For details of the SAL VM interface , please refer to

Document-xxxxx.

2) Generate the interface code using 'salgenerator'

3) Modify the autogenerated sample code to fit the application required.

4) Build if necessary, and test sample programs

**3. Data Definition**

**3.1 Telemetry Definition**

A very simple version of IDL (Interface Definition Language) is used

to define a telemetry topic. The topic is the smallest unit of information which

can be exchanged using the SAL mechanisms.

e.g.

#

# Define the telemetry topic for the IR skycam Application level data

#

#

struct skycam\_IR\_Application {

string<16> site; // none ; none ; Site where instrument is located

long ref\_time; // seconds ; none ; Reference time for this batch of images

string<32> gmt\_time; // none ; none ; GMT version of REF\_TIME

float del\_t; // seconds ; none ; Seconds after REF\_TIME this image began

string<16> htchops; // none ; open|closed|moving|fault ; Hatch position

float encl\_t1; // degC ; -10,30 ; Enclosure internal temperature 1, Celsius

long filpos; // none ; none ; Filter Wheel position

string<16> fildes; // none ; none ; Filter description

float bb\_temps[3]; // degC ; none ; Hatch Blackbody temperatures, Celsius

float fpa\_t; // degC ; -60,0 ; IR camera FPA temperature, Celsius

float duration; // seconds ; none ; Nominal exposure time, sec.

string<128> imagefile; // none ; none ; URL to FITS image

};

This example illustrates the major features :

- comment lines have # in the first column

- all topics are named in a hierarchical fashion intended to

describe their position within the LSST system.

- individual items in a topic are strongly "typed", and may be

of types : string, short, long, float or double. String length is

designated using <nnn> and arrays of other types are

denoted using [nnn].

- Optional comments can be used to define metadata about items,

the format is

// units ; min,max|enumeration ; Brief descriptive text

The following IDL Reserved words may NOT be used in names and will flag an error at

the validation phase (once the SAL System Dictionary is finalized, the item names

will also be validated for compliance with the dictionary).

Reserved words : *bstract any attribute boolean case char component const consumes context custom default double emits enum eventtype exception factory false finder fixed float getraises home import in inout interface local long module multiple native object octet oneway out primarykey private provides public publishes raises readonly sequence setraises short string struct supports switch true truncatable typedef typeid typeprefix union unsigned uses valuebase valuetype void wchar wstring*

**3.2 Command Definition**

The process of defining supported commands is very simple. Commands

are listed (one per line) in a text command\_list file named according to

the subsystem. e.g. command\_list\_dome

### COMMANDS

###-------------------------------------------------------------------------------------------

#type device property action value+modifiers | alias

#

command target position | target

string azimuth

string elevation

command track mode string mode | track

command louvers position set | louvers

double angle[72]

command shutter position open | openShutter

command shutter position close | closeShutter

command target position | park

command target position | movetoCal

command test any any | test

The format of a command definition is

command device attribute action | alias

where value+modifiers are optional and may be primitives (int, string ,etc) or arrays of same.

All alias , subsystem , device, property, action, and names must be present in the SAL System Dictionary. Each value/modifier is defined on a single line and is associated

with the preceding “command” definition.

The command aliases correspond to the ones listed in the relevant subsystem ICD.

3.3 Log Event Definition

Events are defined in a similar fashion to commands. e.g

The file event\_list\_dome contains

### EVENTS

###-------------------------------------------------------------------------------------------

#type id property parameters | alias

###

###

event move ready | slewReady

event move done | slewOK

event move error | slewError

event crawl lock | crawling

event crawl lost | crawlLost

event track lock | tracking

event track lost | trackLost

event louvers done | lldvOK

event louvers error | lldvError

event limit windscreen | screenLimit

event limit jerk | jerkLimit

event limit velocity | VelLimit

event limit acceleration | AccLimit

event limit position | posLimit

string device

string limit

string type

event temperature | tempError

string device

long severity

event power | powerError

string device

long severity

event interlock | interlock

string detail

Optional parameters may be associated with each event, one per line, following

the particular “event” definition.

The Log Event aliases are as defined in the relevant ICD.

**4. Code Generation**

Once Telemetry/Command/Events have been defined , either using the SAL VM or hand edited,

e.g. for ***skycam\_IR\_Application.idl***, interface code and usage samples can

be generated using the ***salgenerator*** tool. e.g.

***salgenerator skycam\_IR\_Application.idl validated***

***salgenerator skycam\_IR\_Application.idl sal cpp***

would generate the c++ communications libraries to be linked with any user code which needs to publish or subscribe to the telemetry stream **skycam\_IR\_Application**.

The item can be wildcarded, so for example

***salgenerator skycam\*.idl sal cpp***

would generate a library appropriate for all skycam related items as well as test programs for each skycam telemetry stream.

The "sal" keyword indicates SAL code generation is the required operation,

the selected wrapper is cpp (GNU G++ compatible code is generated, other options

are java, isocpp and python).

C++ code generation produces a shared library for type support and another

for the SAL API. It also produces test executables to publish and subscribe to all

defined Telemetry streams, and to send all defined Commands and log Events.

Java code generation produces a .jar class library for type support and another

for the SAL API. It also produces .jar libraries to test publishing and subscribing to

all defined Telemetry streams, and to send all defined Commands and log Events.

The Python option generates an import'able library. Simple example scripts to perform the major

functions can be found later in this document.

**4.1 Recommend sequence of operations**

1. Create the IDL and Command and Event definitions

2. Use the salgenerator validate operation

3. Use the salgenerator html operation

4. Use the salgenerator sal operation

5. Verify test programs run correctly

6. Begin simulation/implementation and testing

**4.1.1 Step 1 – Definition**

Use a text editor to create a set of .idl files. Each file should be

appropriately named and consists of a single telemetry stream

definition. The file name should be constructed using the subsystem

name, and a descriptive component. e.g. mount\_TC.idl

**4.1.2 Step 2 – Validation**

Run the salgenerator tool on each .idl file using the validate option.

e.g. salgenerator mount\_TC.idl validate

The successful completion of the validation phase results in the

creation of the following files and directories.

idl-templates – copy of current input id

idl-templates/validated – validated and standardized idl

idl-templates/validated/sal – idl modules for use with OpenSplice

sql – database table definitions for telemetry

xml – XML versions of the telemetry definitions

**4.1.3 Step 3 – Update Strucuture and documentation**

Run the salgenerator tool on each .idl file using the html option

e.g. salgenerator mount\_TC.idl html

The successful completion of the html phase results in the

creation of the following files and directories which may be

used to update the SAL online configuration website. (See SAL VM

documentation for upload details).

html – a set of directories, one per .idl file, with web forms for editing online

a set of index-dbsimulate web page forms

a set of index-simulate web page forms

a set of sal-generator web page forms

**4.1.4 Step 4 – Code Generation**

Run the salgenerator tool on each of the .idl files using the sal option.

The sal option requires at least one target language to also be specified.

The current target languages are cpp, isocpp, java and python.

Depending upon the target language , successful completion of the code

generation results in the following output directories (e.g for mount\_TC)

e.g. salgenerator mount\_TC.idl sal cpp

cpp -

mount: *- common mount support files*

cpp

isocpp

java

mount/cpp:

ccpp\_sal\_mount.h - main include file

libsacpp\_mount\_types.so - dds type support library

Makefile.sacpp\_mount\_types - type support makefile

sal\_mount.cpp - item access support

sal\_mountDcps\_impl.cpp - type class implementation

sal\_mount.idl - type definition idl

sal\_mountDcps.cpp - type support interface

sal\_mountDcps\_impl.h - type implementation headers

sal\_mountSplDcps.cpp - type support I/O

sal\_mountDcps.h - type interface headers

sal\_mount.h - type support class

sal\_mountSplDcps.h - type I/O headers

src

mount/cpp/src:

CheckStatus.cpp - test dds status returns

CheckStatus.h - test dds status headers

mountCommander.cpp - command generator mountController.cpp - command processor

mountEvent.cpp - event generator

mountEventLogger.cpp - event logger

Makefile.sacpp\_mount\_cmd - command support makefile

Makefile.sacpp\_mount\_event - event support makefile

sacpp\_mount\_cmd  *- test program*

sacpp\_mount\_ctl  *- test program*

sacpp\_mount\_event *- test program*

sacpp\_mount\_eventlog  *- test program*

sal\_mount.h - SAL class headers

sal\_mountC.h - SAL C support

sal\_mount.cpp - SAL class

mount\_TC: *- specific to particular telemetry stream*

cpp

isocpp

java

python

mount\_TC/cpp:

src

standalone

mount\_TC/cpp/src:

CheckStatus.cpp - check dds status class

CheckStatus.h - check dds status header

mount\_TCDataPublisher.cpp - Actuators data publisher

mount\_TCDataSubscriber.cpp - Actuators data subscriber

mount\_TC/cpp/standalone:

Makefile

Makefile.sacpp\_mount\_TC\_sub - subscriber makefile

Makefile.sacpp\_mount\_TC\_pub - publisher makefile

sacpp\_mount\_sub  *- test program*

sacpp\_mount\_pub  *- test program*

src

mount\_TC/cpp/standalone/src:

e.g. salgenerator mount\_TC.idl sal isocpp

isocpp -

mount/isocpp:

libISO\_Cxx\_mount\_Typesupport.so - mount support Shared library

Makefile.ISO\_Cxx\_mount\_Typesupport - type support makefile

sal\_mountDcps.h - type support headers

sal\_mount.h - main include file

sal\_mount\_Dcps.hpp - type support classes

sal\_mount.idl - type definition idl

sal\_mount.cpp - SAL mount object class

sal\_mountDcps\_impl.cpp - type support interface classes

sal\_mountSplDcps.cpp - I/O support classes

sal\_mountDcps.cpp - type support classes

sal\_mountDcps\_impl.h - type interface headers

sal\_mountSplDcps.h - I/O support headers

src

mount\_TC/isocpp: *- specific to particular telemetry stream*

implementation.cpp - support classes

implementation.hpp - support headers

Makefile - makefile for test programs

publisher.cpp - publisher source

subscriber.cpp - subscriber source

e.g. salgenerator mount\_TC.idl sal java

java -

mount/java:

classes - compiled type classes

mount - generated java types

Makefile.saj\_mount\_types - makefile fior types

saj\_mount\_types.jar - type support classes

sal\_mount.idl - validated sal idl

src

mount/java/classes:

full set of java .class type support files

mount saj\_mount\_types.manifest

mount/java/classes/mount:

full set of .java type support files

mount/java/mount:

mount/java/src :

ErrorHandler.java

mount\_cmdctl.run - run command tester

mount\_event.run - run event tester

mountCommander.java - commander source

mountController.java - command processor source

mountEvent.java - event generator source

mount\_EventLogger.java - event logger source

Makefile.saj\_mount\_cmdctl - command class makefile

Makefile.saj\_mount\_event - event class makefile

sal\_mount\_cmdctl.jar - command class source

sal\_mount\_event.jar - event class source

mount\_TC/java: *- specific to particular telemetry stream*

Makefile

src

standalone

mount\_TC/java/src:

ErrorHandler.java - error handler class source

mount\_TCDataPublisher.java - publisher class source

mount\_TCDataSubscriber.java - subscriber class source

org

mount\_TC/java/src/org:

lsst

mount\_TC/java/src/org/lsst:

sal

mount\_TC/java/src/org/lsst/sal:

sal\_mount.java - sal class for mount

mount\_TC/java/src/org/lsst/sal/mount:

Actuators

mount\_TC/java/src/org/lsst/sal/mount/Actuators:

mount\_TC/java/standalone:

mount\_TC.run *- run test programs*

Makefile

Makefile.saj\_mount\_TC\_pub - publication class makefile Makefile.saj\_mount\_TC\_sub - subscription class makefile

saj\_mount\_TC\_pub.jar - telemetry publication class

saj\_mount\_TC\_sub.jar - telemetry subscription class

e.g. salgenerator mount\_TC.idl sal python

mount/cpp/src :

Makefile\_sacpp\_mount\_python

SALPY\_mount.cpp - Boost.python wrapper

SALPY\_mount.so - import'able python library

4.1.5 **Step 5 – Verification , testing and Integration**

The default OpenSplice configuration requires that certain firewall rules are added,

alternatively, shut down the firewall whilst testing.

For iptables : this can be done (as root) with the following commands

*/etc/init.d/iptables stop*

or by editing the

*/etc/sysconfig/iptables*

to add the following lines

*-A INPUT -p udp -m udp --dport 250:251 -j ACCEPT*

*-A INPUT -p udp -m udp --dport 7400:7411 -j ACCEPT*

*-A OUTPUT -p udp -m udp --dport 250:251 -j ACCEPT*

*-A OUTPUT -p udp -m udp --dport 7400:7411 -j ACCEPT*

The iptables service should then be restarted

*/etc/init.d/iptables restarted*

For firewalld : this can be done (as root) with the following commands

First, run the following command to find the default zone:

*firewall-cmd --get-default-zone*

Next, issue the following commands:

*firewall-cmd --zone=public --add-port=250-251/udp --permanent*

*firewall-cmd --zone=public --add-port=7400-7411/udp --permanent*

*firewall-cmd --reload*

Replace public with whatever the default zone says if it is different.

**4.2 salgenerator Options**

The salgenerator executes a variety of processes, depending upon the options

selected.

validate - check the .idl files, , command\_list and event\_list

html - generate web form interfaces and documentation

labview - generate Labview interface

sal [lang] - generate SAL C++, Java, or Python wrappers

simd - generate simd wrappers (deprecated)

shmem - generate shared memory interface

sim - generate simulation configuration

tcl - generate tcl interface

icd - generate ICD document

maven - generate a maven project (per subsystem)

verbose - be more verbose ;-)

db - generate telemetry database table

for db the arguments required are

db start-time end-time interval

where the times are formatted like “2008-11-12 16:20:01“

and the interval is in seconds

**4.3 SAL examples**

**4.3.1 Example 1 – Publishing telemetry**

**Using C++**

mount\_TCC myData;

long i,iseq;

SAL\_mount mgr = SAL\_Mount();

//create publisher

mgr.salTelemetryPub(“mount\_TC”);

//set data values

for (I-0;i<18;i++) {myData.Raw[i] = i;}

for (I-0;i<18;i++) {myData.Calibrated[i] = i;}

//publish the sample

mgr.putSample\_TC(&myData);

//tidyup

mgr.salShutdown();

**Using Java**

// initialize

SAL\_mount mgr = SAL\_mount();

// create publisher

mgr.salTelemetryPub("mount\_TC");

Actuators myData = new Actuators();

//set data values

for (int i-0;i<18;i++) {myData.Raw[i] = i;}

for (int i-0;i<18;i++) {myData.Calibrated[i] = i;}

//publish the sample

mgr.putSample(myData);

//tidyup

mgr.salShutdown();

**Using Python**

# initialize

from SALPY\_mount import \*

mgr=SAL\_mount()

myData=mount\_TC()

# create publisher

mgr.salTelemetryPub(“mount\_TC”)

#set data values

for i in range (0,18)

myData.Raw[i]=i

myData.Calibrated[i]=i

# publish the sample

mgr.putTC(myData)

# tidyup

mgr.salShutdown()

**4.3.2 Example 2 – Sending a command**

**Using C++**

// initialize

SAL\_mount cmd = SAL\_mount();

// create command object

cmd.salCommand();

mount::command command; /\* Example on Stack \*/

command.device = DDS::string\_dup(device);

command.property = DDS::string\_dup(property);

command.action = DDS::string\_dup(action);

command.value = DDS::string\_dup(value);

command.modifiers = DDS::string\_dup(modifiers);

// send the command

cmdId = cmd.issueCommand(command);

// wait for ack/completion

os\_nanoSleep(delay\_1s);

status = cmd.waitForCompletion(cmdId, timeout);

// tidyup

cmd.salShutdown();

**Using Java**

// initialize

SAL\_mount mgr = new SAL\_mount();

// Issue command

int cmdId=1;

int timeout=5; //seconds

int status=0;

// create command object

mgr.salCommand();

static command = new mount.command();

command.device = “rotator”;

command.property = “angle”;

command.action = “move”;

command.value = “23.0”

command.modifiers = “”;

cmdId = mgr.issueCommand(command);

Thread.sleep(1000);

status = mgr.waitForCompletion(cmdId, timeout);

// tidyup

mgr.salShutdown();

**Using Python**

# intialize

from SALPY\_mount import \*

mgr=SAL\_mount(1)

# create command object

mgr.salCommand()

command=mount\_CommandC()

command.device=”rotater”

command.property=”angle”

command.action=”move”

command.value=”23.0”

command.modifiers=””

# send the command

cmdId=mgr.issueCommand(command)

# wait for ack/completion

status=cmd.waitForCompletion(cmdId)

# tidyup

mgr.salShutdown()

**4.3.3 Example 3 – Receiving commands**

**Using C++**

// initialize

int timeout=5;

SAL\_mount cmd = SAL\_mount();

// create command object

cmd.salProcessor();

mount::commandSeq command; /\* Example on Stack \*/

// wait for a command to arrive

cmdId = cmd.acceptCommand(command);

if (cmdId > 0) {

if (timeout > 0) {

// take some time to complete

cmd.ackCommand(cmdId, SAL\_\_CMD\_INPROGRESS, timeout, "Ack : OK");

os\_nanoSleep(delay);

}

// pass back command completion ack

cmd.ackCommand(cmdId, SAL\_\_CMD\_COMPLETE, 0, "Done : OK");

}

// tiduyp

cmd.salShutdown();

**Using Java**

// initialize

SAL\_mount cmd = new SAL\_mount();

int status = SAL\_\_OK;

int cmdId = 0;

int timeout = 0;

// Initialize

cmd.salProcessor();

command = new mount::commandSeq();

// wait for command to arrive

cmdId = cmd.acceptCommand(command);

if (cmdId > 0) {

if (timeout > 0) {

// take some time to complete

cmd.ackCommand(cmdId, SAL\_\_CMD\_INPROGRESS, timeout, "Ack : OK");

Thread.sleep(timeout);

}

// pass back command completion ack

cmd.ackCommand(cmdId, SAL\_\_CMD\_COMPLETE, 0, "Done : OK");

}

// tidyup

cmd.salShutdown();

**Using Python**

# initialize

from SALPY\_mount import \*

cmd=SAL\_mount()

cmd.salProcessor()

command=mount\_CommandC()

// wait for a command to arrive

cmdId=cmd.acceptCommand(command)

// pass back command completion ack

cmd.ackCommand(cmdId,SAL\_\_OK , 0 , ”OK”)

// tiduyp

cmd.salShutdown()

**4.3.4 Example 4 – Generating an Event**

**Using C++**

int priority = SAL\_\_EVENT\_INFO;

SAL\_mount mgr = SAL\_mount();

string message=”Testing the Event mechanism”;

// generate event

mgr.logEvent(message.c\_str(), priority);

cout << "=== Event " << alias << " generated = " << message << endl;

// tidyup

mgr.salShutdown();

**Using Java**

// Initialize

int status=0;

SAL\_mount mgr = new SAL\_mount();

String msg=”Testing the Event mechanism”;

int priority=1;

status = mgr.logEvent(msg,priority);

// tidyup

mgr.salShutdown();

**Using Python**

from SALPY\_mount import \*

h=SAL\_mount(1)

h.logEvent(“Testing the Event mechanism”,1)

5. Testing

5.1 Environment

To check that the OpenSplice environment has been correctly initialized ; in a terminal, type

ipcs -a

(lists shared memory segments)

idlpp

(tests availability of idl processor)

To check that the SAL environment has been correctly initialized; in a terminal type

salgenerator

(tests availability of sal processor/generator)

5.2 Telemetry

Once the salgenerator has been used to validate the definition files and generate the

support libraries, there will be automatically built test programs available.

In all cases , log and diagnostic output from OpenSplice will be written to the

files

ospl-info.log and ospl-error.log

in the directory where the test is run.

The following locations assume code has been built for the skycam subsystem support,

there will be separate subdirectories for each Telemetry stream type.

For C++

skycam\_<telemetryType>/cpp/standalone/sacpp\_skycam\_<telemetryType>\_pub - publisher

skycam\_<telemetryType>/cpp/standalone/sacpp\_skycam\_<telemetryType>\_sub - subscriber

For java

skycam\_<telemetryType>/java/standalone/skycam\_<telemetryType>.run

- start publisher and subscriber

5.3 Commands

The following locations assume code has been built for mount subsystem support

For C++

mount/cpp/src/sacpp\_mount\_cmd - to send commands

mount/cpp/src/sacpp\_mount\_ctrl - to process commands

For java

mount/java/src/mount\_cmdctl.run - starts command processor

In addition a gui can be used to send all supported subsystem commands (with am associated

processor to demonstrate reception of same). To start the gui e.g. for hexapod subsystem

For C++

command\_test\_gui hexapod

The gui provides a window to select the command to run. If a command has optional values

/modifiers, then a subwindow will open to allow their values to be entered.

A terminal window show the messages from a demo command processor which simply

prints the contents of commands as they are received.



5.4 Events

The following locations assume code has been built for mount subsystem support

For C++

mount/cpp/src/sacpp\_mount\_event - to generate events

mount/cpp/src/sacpp\_mount\_eventlog - to log the events

For java

mount/java/src/mount\_events.run - starts events processor

In addition a gui can be used to send all supported subsystem commands (with an associated

processor to demonstrate reception of same). To start the gui e.g. for hexapod subsystem

For C++

logevent\_test\_gui hexapod

The gui provides a window to select the event to generate.. If an event has optional values

/modifiers, then a subwindow will open to allow their values to be entered.

A terminal window show the messages from a demo event processor which simply

prints the contents of events as they are received.



5.5 TCS pointing simulator

The SDK includes a TCS pointing kernel simulation, with associated gui's and data files.

This can be found in the

*/opt/test/tcs/tcs*

directory tree.

The simulation consists of the following elements, all of which communicate using the

SAL layer (C++).

a). TCS pointing kernel with GUI and command line

b). Opsim database log , used as input

c). Mount controller simulator

d). Camera controller simulator

e). Hexapod controller simulators

f). Dome controller simulator

g). Rotator controller simulator

h). M2 controller simulator

The simulation is started by

*cd /opt/test/tcs/tcs/bin*

*./startdemo*

Once all the windows have deployed, the tcs simulator will automatically slew to the default

target. Once it arrives (watch the GUI to follow it's progress), locate the command line interface window and type

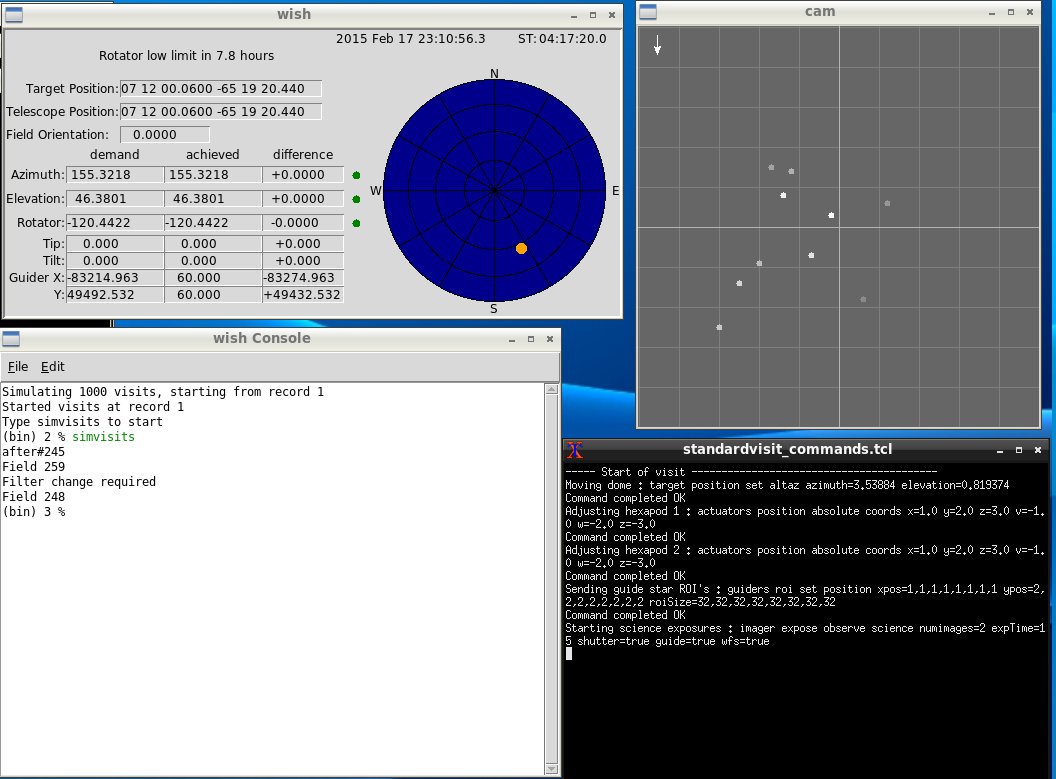
*simvisits*

to start the simulated set of visits.

For each new visit, the simulator will send appropriately timed commands to each of

the subsystem controller simulators.

TCS Simulation GUI



Standard Visit window



Simulated Subsystem Controllers



Customized controller simulators can also be used by specifying their location via

environment variables

e.g.

export LSST\_DOME\_SIMULATOR /home/saldev/bin/dome\_controller\_test

would change any subsequent “startdemo” invocations to use the specified executable

for the dome controller instead of the default one.

6. Application programming Interfaces

6.1. C++

Includes :

#include <string>

#include <sstream>

#include <iostream>

#include "SAL\_mount.h"

#include "ccpp\_sal\_mount.h"

#include "os.h"

#include "example\_main.h"

using namespace DDS;

using namespace <subssytem>; // substitute the actual subsystem name here

Public :

int putSample(<subsystem::telemetryType> data); - publish telemetry sample

int getSample(<subsystem::telemetryTypeSeq> data); - read next telemetry sample

int putSample\_<telemetryType>( <subsystem::telemetryTypeC>\*data); - publish telemetry sample (C)

int getSample\_<telemetryType>(<subsystem::telemetryTypeC>\*data); - read next telemetry sample (C)

void salTypeSupport(char \*topicName); - initialize type support

void salTelemetryPub(char \*topicName); - create telemetry publishber

void salTelemetrySub(char \*topicName); - create telemetry subscriber

void salEvent(char \*topicName); - create event object

int getResponse(<subsystem>::ackcmdSeq data); - read command ack

int getEvent(<subsystem>::logeventSeq data); - read event data

void salShutdown(); - tidyup

void salCommand(); - create command object

void salProcessor(); - create command processor object

int issueCommand( <subsystem>::command data); - send a command

int issueCommandC( <subsystem>\_commandC \*data); - send a command (C)

int ackCommand( int cmdSeqNum, long ack, - acknowledge a command

long error, char \*result );

int acceptCommand( <subsystem>::commandSeq data); - read next command

int acceptCommandC( <subsystem>\_commandC \*data); - read next command (C)

int checkCommand( int cmdSeqNum ); - check command status

int cancelCommand( int cmdSeqNum ); - cancel command

int abortCommand( int cmdSeqNum ); - abort all commands

int waitForCompletion( int cmdSeqNum ,unsigned int timeout ); - wait for command to complete

int setDebugLevel( int level ); - change debug info level

int getDebugLevel( int level ); - get current debug info level

int getOrigin(); - get origin descriptor

int getProperty(stringproperty, stringvalue); - get configuration item

int setProperty(stringproperty, stringvalue); - set configuration item

int getPolicy(stringpolicy, stringvalue); - get middleware policy item

int setPolicy(stringpolicy, stringvalue); - set middleware policy item

void logError(int status); - log middleware error

salTIME currentTime(); - get current timestamp

int logEvent( char \*message, int priority ); - generate a log event

6.2 Java

Includes :

import <subsystem>.\*; //substitute actual subsystem name here

import org.lsst.sal.<SAL\_subsystem>; //substitute actual subsystem name here

Public :

public void salTypeSupport(String topicName) - initialize type support

public int putSample(<telemetryType> data) - publish a telemetry sample

public int getSample(<telemetryType> data) - read next telemetry sample

public void salTelemetryPub(String topicName) - create telemetry publisher

public void salTelemetrySub(String topicName) - create telemetry subscriber

public void logError(int status) - log middleware error

public SAL\_<subsystem>() - create SAL object

public int issueCommand( command data ) - send a command

public int ackCommand( int cmdId, int ack, int error, String result ) - acknowledge a command

public int acceptCommand( <subsystem>.command data ) - read next command

public int checkCommand( int cmdSeqNum ) - check command status

public int getResponse(ackcmdSeqHolder data) - read command ack

public int cancelCommand( int cmdSeqNum ) - cancel a command

public int abortCommand( int cmdSeqNum ) - abort all commands

public int waitForCompletion( int cmdSeqNum , int timeout ) - wait for command to complete

public int getEvent(logeventSeqHolder data) - read next event data

public int logEvent( String message, int priority ) - generate an event

public int setDebugLevel( int level ) - set debug info level

public int getDebugLevel( int level ) - get debug info level

public int getOrigin() - get origin descriptor

public int getProperty(String property, String value) - get configuration item

public int setProperty(String property, String value) - set configuration item

public void salCommand() - create a command object

public void salProcessor() - create command processor object

public void salShutdown() - tidyup

public void salEvent(String topicName) - create event object

6.3 Python (Boost.python bindings)

BOOST\_PYTHON\_MODULE(SALPY\_mount){

namespace bp = boost::python;

bp::class\_<subsystem\_TelemetryTypeC>("subsystem\_TelemetryTypeC")

.add\_property("telemetryItem", make\_array(&<subsystem::TelemetryTypeC>::telemetryItem))

bp::class\_<SAL\_subsystem>("SAL\_subsystem", bp::init<int>())

.def(bp::init<int>())

.def(

"abortCommand"

, (::int ( ::SAL\_subsystem::\* )( int ) )( &::SAL\_subsystem::abortCommand )

, ( bp::arg("cmdSeqNum") ) )

.def(

"acceptCommand"

, (::int ( ::SAL\_subsystem::\* )( ::mount\_commandC ) )( &::SAL\_subsystem::acceptCommandC )

, ( bp::arg("data") ) )

.def(

"ackCommand"

, (::int ( ::SAL\_subsystem::\* )( int,::long,::long,char \* ) )( &::SAL\_subsystem::ackCommand )

, ( bp::arg("cmdSeqNum"), bp::arg("ack"), bp::arg("error"), bp::arg("result") ) )

.def(

"cancelCommand"

, (::int ( ::SAL\_subsystem::\* )( int ) )( &::SAL\_subsystem::cancelCommand )

, ( bp::arg("cmdSeqNum") ) )

.def(

"checkCommand"

, (::int ( ::SAL\_subsystem::\* )( int ) )( &::SAL\_subsystem::checkCommand )

, ( bp::arg("cmdSeqNum") ) )

.def(

"currentTime"

, (::salTIME ( ::SAL\_subsystem::\* )( ) )( &::SAL\_subsystem::currentTime ) )

.def(

"getDebugLevel"

, (int ( ::SAL\_subsystem::\* )( int ) )( &::SAL\_subsystem::getDebugLevel )

, ( bp::arg("level") ) )

.def(

"getEvent"

, (::int ( ::SAL\_subsystem::\* )( ::subsystem\_logeventC ) )( &::SAL\_subsystem::getEvent )

, ( bp::arg("data") ) )

.def(

"getOrigin"

, (int ( ::SAL\_subsystem::\* )( ) )( &::SAL\_subsystem::getOrigin ) )

.def(

"getProperty"

, (int ( ::SAL\_subsystem::\* )( char \*,char \* ) )( &::SAL\_subsystem::getProperty )

, ( bp::arg("property"), bp::arg("value") ) )

.def(

"getResponse"

, (::int ( ::SAL\_subsystem::\* )( ::subsystem\_ackcmdC ) )( &::SAL\_subsystem::getResponse )

, ( bp::arg("data") ) )

.def(

"issueCommand"

, (int ( ::SAL\_subsystem::\* )( ::subssytem\_commandC ) )( &::SAL\_subsystem::issueCommandC )

, ( bp::arg("data") ) )

.def(

"logError"

, (void ( ::SAL\_subsystem::\* )( ::int ) )( &::SAL\_subsystem::logError )

, ( bp::arg("status") ) )

.def(

"logEvent"

, (::int ( ::SAL\_subsystem::\* )( char \*,int ) )( &::SAL\_subsystem::logEvent )

, ( bp::arg("message"), bp::arg("priority") ) )

.def(

"salCommand"

, (void ( ::SAL\_subsystem::\* )( ) )( &::SAL\_subsystem::salCommand ) )

.def(

"salProcessor"

, (void ( ::SAL\_subsystem::\* )( ) )( &::SAL\_subsystem::salProcessor ) )

.def(

"salShutdown"

, (void ( ::SAL\_subsystem::\* )( ) )( &::SAL\_subsystem::salShutdown ) )

.def(

"salTelemetryPub"

, (void ( ::SAL\_subsystem::\* )( char \* ) )( &::SAL\_subsystem::salTelemetryPub )

, ( bp::arg("topicName") ) )

.def(

"salTelemetrySub"

, (void ( ::SAL\_subsystem::\* )( char \* ) )( &::SAL\_subsystem::salTelemetrySub )

, ( bp::arg("topicName") ) )

.def(

"salTypeSupport"

, (void ( ::SAL\_subsystem::\* )( char \* ) )( &::SAL\_subsystem::salTypeSupport )

, ( bp::arg("topicName") ) )

.def(

"setDebugLevel"

, (::int ( ::SAL\_subsystem::\* )( int ) )( &::SAL\_subsystem::setDebugLevel )

, ( bp::arg("level") ) )

.def(

"setProperty"

, (::int ( ::SAL\_subsystem::\* )( char \*,char \* ) )( &::SAL\_subsystem::setProperty )

, ( bp::arg("property"), bp::arg("value") ) )

.def(

"waitForCompletion"

, (::int ( ::SAL\_subsystem::\* )( int,int ) )( &::SAL\_subsystem::waitForCompletion )

, ( bp::arg("cmdSeqNum"), bp::arg("timeout") ) )

.def(

"get<TelemetryType" , &::SAL\_subsystem::<getSampleTelemetryType> )

.def(

"put<TelemetryType" , &::SAL\_subsystem::<putSampleTelemetryType> )

bp::class\_< subsystem\_ackcmdC >( "subsystem\_ackcmdC" )

.def\_readwrite( "ack", &subsystem\_ackcmdC::ack )

.def\_readwrite( "error", &subsystem\_ackcmdC::error )

.def\_readwrite( "result", &subsystem\_ackcmdC::result )

;

bp::class\_< subsystem\_commandC >( "subsystem\_commandC" )

.def\_readwrite( "device", &usbsystem\_commandC::device )

.def\_readwrite( "property", &subsystem\_commandC::property )

.def\_readwrite( "action", &subsystem\_commandC::action )

.def\_readwrite( "value", &subsystem\_commandC::value )

.def\_readwrite( "modifiers", &subsystem\_commandC::modifiers )

;

bp::class\_< subsystem\_logeventC >( "subssytem\_logeventC" )

.def\_readwrite( "message", &subsystem\_logeventC::message )

;