

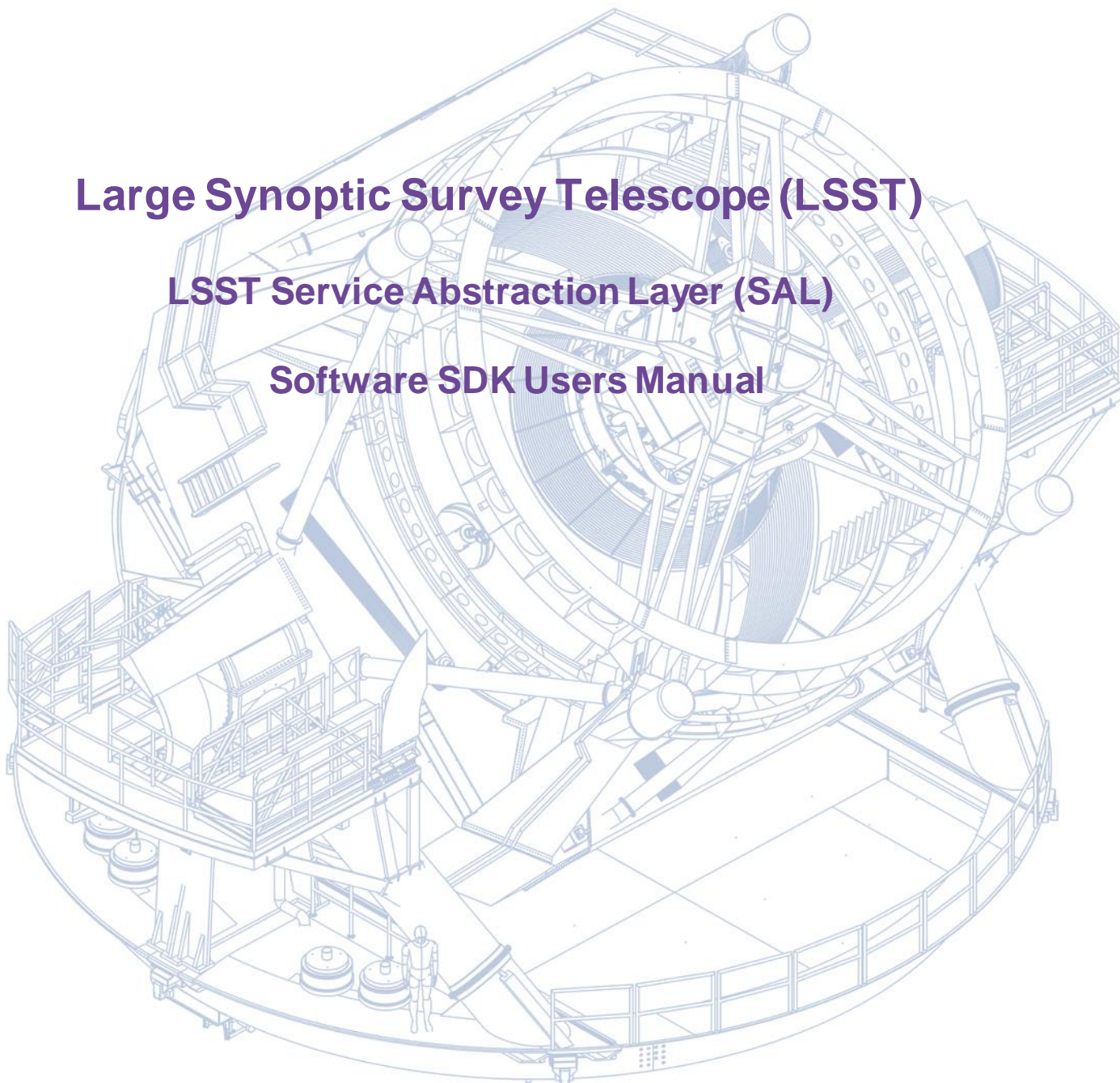


LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)

LSST Service Abstraction Layer (SAL)

Software SDK Users Manual



Change Record

Version	Date	Description	Owner name
V1.0	12/13/2013	Initial Draft	D. Mills
V2.0	06/06/2014	First Release	D. Mills
V2.3	06/05/2015	Added detail about large-file handling	D. Mills
V2.4	07/27/2015	Major revision to document XML schema use	D. Mills
V3.0	04/29/2016	Revisions for SAL V3 release	D. Mills
V3.5	02/23/2016	Revisions for SAL V3.5 release	D.Mills
V3.6	03/09/2018	Add lists of generic commands and events	D. Mills

Table of Contents

LSST Service Abstraction Layer (SAL)	1
Software SDK Users Manual	1
1 Introduction	5
2. Installation	6
2.0.1 Installation from a tar archive release	6
2.0.2 Installation from Git repositories	7
2.0.3 Install location customization	7
2.1 Installation in a virtual machine	8
2.2 Standard usage	9
3. Data Definition	9
3.1 Telemetry Definition	10
3.2 Command Definition	11
3.3 Log Event Definition	12
3.4 Updating the XML definitions	13
4. Using the SDK	14
4.1 Recommend sequence of operations	15
4.1.1 Step 1 – Definition	16
4.1.2 Step 2 – Validation	16
4.1.3 Step 3 – Update Structure and documentation	16
4.1.4 Step 4 – Code Generation	17
4.2 salgenerator Options	22
4.3 SAL API examples	23
5. Testing	24
5.1 Environment	24
5.2 Telemetry	25
5.3 Commands	25
	27
5.4 Events	28

6. Application programming Interfaces	29
6.1. C++	29
6.2 Java	30
6.3 Python (pybind11 bindings)	31
7.0 SAL XML Schema	32
7.1 Telemetry	32
7.1.1 telemetrySetType	32
	32
7.1.2 telemetryType	32
	32
7.1.3 telemetryItemType	33
	33
7.2 Commands	34
7.2.1 commandSetType	34
	34
7.2.2 commandType	34
	34
7.2.3 commandItemType	35
	35
7.3 Events	36
7.3.1 eventSetType	36
7.3.2 eventType	36
7.3.3 eventItemType	37
8.0 Compiler Options and Link Libraries	38

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 Events).

The SAL SDK should be installed on a modern (x86_64) Linux computer. The current baseline recommended configuration is 64-bit CentOS 7.3.

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). Installation of system packages must be done using sudo (eg sudo yum install , or sudo apt-get install).

- gcc-c++
- make
- ncurses-libs or ncurses-dev
- xterm
- xorg-x11-fonts-misc
- java-1.7.0-openjdk-devel
- maven
- python-devel
- swig
- git
- tk-devel

The distribution includes dedicated versions of the following packages

- OpenSplice

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:///path-to-installation/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.

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 command
`sudo /etc/init.d/iptables stop`

For Ubuntu: use `sudo ufw disable`

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=7400-7413/udp --permanent
```

```
firewall-cmd --reload
```

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

The location of the OpenSplice configuration file is stored in the environment variable `OSPL_URI`, and an extensive configuration tool exists (*osplconf*), should customization be necessary.

2.0.1 Installation from a tar archive release

The tar archive format release includes a compatible version of OpenSplice as well as the SAL toolkit.

Unpack the SAL tar archive in a location of choice (/opt is recommended), e.g. in a terminal, replacing x.y.z with the appropriate version id

```
cd /opt
```

```
tar xzf [location-of-sdk-archive]/saSDK-x.y.z_x86_64.tgz
```

and then add the SDK setup command.

```
source /opt/setup.env
```

to your bash login profile.

2.0.2 Installation from Git repositories

Use a git client of your preference to check out the required branch of the following repositories

https://github.com/lsst-ts/ts_sal
https://github.com/lsst-ts/ts_opensplice

and then add the SDK setup command.

```
source /opt/setup.env
```

to your bash login profile.

2.0.3 Install location customization

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 standard location for the OpenSplice package is in the same directory as the SDK, But you can install it elsewhere as long as you edit the OSPL_HOME environment variable to reference the actual path.

Another important environment variable is SAL_WORK_DIR. This is the directory in which you will run the SAL tools, and in which all the output files and libraries will be generated. By default this will be the “test” subdirectory in LSST_SDK_INSTALL, but you can change SAL_WORK_DIR to redefine it if required.

ALL THE salgenerator STEPS MUST BE RUN FROM THE SAL_WORK_DIR DIRECTORY

If you will be running SAL applications in parallel with other users on your subnet, it is advisable to partition your network traffic so as not to interfere with each others activities. This can be done by setting the environment variable LSST_DDS_DOMAIN to a unique string value for each user.

Also retrieve ts_xml and copy the appropriate subsystem definitions to your working directory.
e.g. `cp ts_xml-master/sal_interfaces/mysubsystem/*.xml test/.`

Where test is the working directory specified by the SAL_WORK_DIR environment variable.

Add the invocation of setup.env to your bash login profile

```
source /sal-install-directory-path/setup.env
```

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.

The current prototypes for each subsystem can be used as a baseline, eg for the dome subsystem

```
cd $SAL_WORK_DIR
cp $SAL_HOME/scripts/xml-templates/dome/*.xml .
```

- 2) Generate the interface code using 'salgenerator'
- 3) Modify the autogenerated sample code to fit the application required.
- 4) Build if necessary, and test the sample programs

Example makefiles are provided for all the test programs. The list of libraries required to link with the middleware can be found in section 8.0

2.1 Installation in a virtual machine

The SDK has been tested in a Virtual Machine environment (VirtualBox).

To set up a VM appropriately for this usage :

1. In VM configuration , choose Bridged Adaptor for the network device
2. Add a sal user account during OS installation, the user should be an administrator
3. Choose Gnome Desktop + Development tools during OS installation
4. From VM menu , install Guest Additions

5. Once the OS has booted, enable the network
6. Verify the network is ok.
7. `sudo yum install xterm xorg-x11-fonts-misc java-1.7.0-openjdk-devel boost-python boost-python-devel maven python-devel tk-devel`
8. Configure (or disable) iptables and firewalld
eg `systemctl disable iptables`
`systemctl disable firewalld`
`system stop iptables`
`system stop firewalld`

2.2 Standard usage

Normal usage of the SDK comprise four main steps

1. Define Telemetry, Command, and Event datatypes (either using the SAL VM website interface, or an asci or XML editor). In some cases the XML from another subsystem might provide a useful bootstrap. See the `ts_xml` repository.
2. Generate the interface code using the 'salgenerator'
3. Modify the autogenerated sample code to fit the application required
4. Build and test the sample programs

Example makefiles are provided for all the test programs. The list of libraries required to link an application with the middleware can be found in section 8.0

3. Data Definition

In all XML data definition files the `IDL_Type` keyword is used to specify the datatype of each field. The following datatypes are supported:

- short
- long (this is a 4 byte integer, and is represented as `int` on Linux 64-bit)
- long long (8 byte integer)
- unsigned short
- unsigned long (this is a 4 byte integer, and is represented as `int` on Linux 64-bit)

- unsigned long long (8 byte integer)
- float
- double
- char , specify length using the Count tag
- boolean
- octet (sequence of unsigned bytes)
- string, specify length using the Count tag
- numeric arrays, use the Count tag with any numeric type

If there is a time-of-data associated with an item, then it should be named “*timestamp*”,

and be of type *double*. The time should be TAI time as returned by the *getCurrentTime* method. If more than one timestamp is needed in a topic, then they should be named

“*timestamp-name1, timestamp-name2* etc”. If an array of times is required, then the type should be “*double timestamp[size]*”.

3.1 Telemetry Definition

A very simple XML schema is used to define a telemetry topic.

The topic is the smallest unit of information which can be exchanged using the SAL mechanisms.

The following 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 dec default double emits enum eventtype exception factory false finder fixed float getraises home import in inout interface limit 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*

e.g.

```
<SALTelemetry>
<Subsystem>hexapod</Subsystem>
<Version>2.5</Version>
<Author>A Developer</Author>
<EFDB_Topic>hexapod_LimitSensors</EFDB_Topic>
  <item>
    <EFDB_Name>liftoff</EFDB_Name>
    <Description></Description>
    <Frequency>0.054</Frequency>
    <IDL_Type>short</IDL_Type>
    <Units></Units>
    <Conversion></Conversion>
    <Count>18</Count>
  </item>
  <item>
    <EFDB_Name>limit</EFDB_Name>
    <Description></Description>
    <Frequency>0.054</Frequency>
    <IDL_Type>short</IDL_Type>
    <Units></Units>
    <Count>18</Count>
  </item>
</SALTelemetry>
```

3.2 Command Definition

The process of defining supported commands is similar to Telemetry using XML.
The command aliases correspond to the ones listed in the relevant subsystem ICD.
e.g.

```
<SALCommand>
  <Subsystem>hexapod</Subsystem>
  <Version>2.5</Version>
  <Author>salgenerator</Author>
  <EFDB_Topic>hexapod_command_configureAcceleration</EFDB_Topic>
  <Alias>configureAcceleration</Alias>
  <Device>drive</Device>
  <Property>acceleration</Property>
  <Action></Action>
  <Value></Value>
  <Explanation>http://sal.sst.org/SAL/Commands/hexapod\_command\_configureAcceleration.html</Explan
ation>
  <item>
    <EFDB_Name>xmin</EFDB_Name>
    <Description> </Description>
    <IDL_Type>double</IDL_Type>
    <Units> </Units>
    <Count>1</Count>
  </item>
  <item>
    <EFDB_Name>xmax</EFDB_Name>
    <Description> </Description>
    <IDL_Type>double</IDL_Type>
    <Units> </Units>
    <Count>1</Count>
  </item>
  <item>
  </item>
</SALCommand>
```

Note : The generic lifecycle commands should NOT be included , they are automatically generated during the salgenerator validation and/or UML to XML processing. The current generic command set is {*start, stop, enable, disable, abort, enterControl, exitControl, standby, SetValue* }

3.3 Log Event Definition

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

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

e.g.

```

<SALEvent>
  <Subsystem>hexapod</Subsystem>
  <Version>2.4</Version>
  <Author>salgenerator</Author>
  <EFDB_Topic>hexapod_logevent_limit</EFDB_Topic>
  <Alias>limit</Alias>
  <Explanation>http://sal.lsst.org/SAL/Events/hexapod_logevent_limit.html</Explanation>
  <item>
    <EFDB_Name>priority</EFDB_Name>
    <Description>Severity of the event</Description>
    <IDL_Type>long</IDL_Type>
    <Units>NA</Units>
    <Count>1</Count>
  </item>
  <item>
    <EFDB_Name>axis</EFDB_Name>
    <Description> </Description>
    <IDL_Type>string</IDL_Type>
    <Units> </Units>
    <Count>1</Count>
  </item>
  <item>
    <EFDB_Name>limit</EFDB_Name>
    <Description> </Description>
    <IDL_Type>string</IDL_Type>
    <Units> </Units>
    <Count>1</Count>
  </item>
  <item>
    <EFDB_Name>type</EFDB_Name>
    <Description> </Description>
    <IDL_Type>string</IDL_Type>
    <Units> </Units>
    <Count>1</Count>
  </item>
</SALEvent>

```

Note : The generic lifecycle events should NOT be included , they are automatically generated during the salgenerator validation and/or UML to XML processing. The current generic event set is *SettingVersions*, *SummaryState*, *DetailedState*, *ErrorCode*, *AppliedSettingsMatchStart*.

3.4 Updating the XML definitions

The XML definitions of the SAL objects for each subsystem are maintained in a github repository (https://github.com/lsst-ts/ts_xml).

When subsystem developers update the XML definitions for their interfaces, they should create a new feature branch in the github repository and put the modified version into it. Once the feature(s) have been fully tested, the corresponding changes are made to the appropriate ICD. Once the ICD has been approved by the Change Control Board, the modified XML will be merged into the master branch and assigned an official release number. The master (release) branch is used to generate the SAL runtime libraries which can be used by other subsystems for integration testing. The master branch is also used by the Continuous Integration Unit Testing framework.

The XML definition files for the subsystem you are developing should be checked out of the github repository to ensure you are working with the latest version.

For convenience the full set of current definition files is also included in each SAL SDK Release (in `lstsal/scripts/xml-templates`).

The XML definition files should be copied to the `SAL_WORK_DIR` directory before using the SAL tools.

The SAL tools must be run from the `SAL_WORK_DIR` directory.

4. Using the SDK

Before using the SDK, make sure that all the directories in the `SAL_WORK_DIR` and The SAL installation directory are owned by you

e.g.

```
cd $SAL_WORK_DIR
chown -R <username>:<username> *
```

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

e.g. for *skycam*, interface code and usage samples can be generated using the *salgenerator* tool. e.g.

```
salgenerator skycam validate
salgenerator skycam sal cpp
```

would generate the c++ communications libraries to be linked with any user code which needs to interface with the **skycam** subsystem.

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 importable library. Simple example scripts to perform the major functions can be found later in this document.

The "labview" keyword indicates that a LabVIEW compatible shared library and Monitor task should be built (the "sal cpp" step must previously have been run).

The "maven" keyword indicates that a Maven project should be built for the subsystem. This will be placed in \$SAL_WORK_DIR/maven/[subsystem]_[version], The "sal java" step must previously have been run).

4.1 Recommend sequence of operations

1. Create the XML Telemetry , 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. Build the SAL shared library / JAR for the subsystem
7. Begin simulation/implementation and testing

4.1.1 Step 1 – Definition

Use an XML editor to create/modify the set of subsystem xml files. Each file should be appropriately named and consists of a either Telemetry, Command, or Event definitions. The current prototypes for each subsystem can be found at https://github.com/lst-ts/ts_xml.

4.1.2 Step 2 – Validation

Run the salgenerator tool validate option for the appropriate subsystem.

e.g. `salgenerator mount validate`

The successful completion of the validation phase results in the creation of the following files and directories.

- idl-templates – Corresponding IDL DDS topic definitions
- 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 all telemetry definitions

4.1.3 Step 3 – Update Structure and documentation

Run the salgenerator html option for the appropriate subsystem.

e.g. `salgenerator mount 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 using the sal option for the appropriate subsystem.
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)

e.g. salgenerator mount 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
sal_mount.h
sal_mountSplDcps.h
src

- type interface headers
- type support class
- type I/O headers

mount/cpp/src:

CheckStatus.cpp
CheckStatus.h
mountCommander.cpp
mountController.cpp
mountEvent.cpp
mountEventLogger.cpp
Makefile.sacpp_mount_cmd
Makefile.sacpp_mount_event
sacpp_mount_cmd
sacpp_mount_ctl
sacpp_mount_event
sacpp_mount_eventlog
sal_mount.h
sal_mountC.h
sal_mount.cpp

- test dds status returns
- test dds status headers
- command generator
- command processor
- event generator
- event logger
- command support makefile
- event support makefile
- *test program*
- *test program*
- *test program*
- *test program*
- SAL class headers
- SAL C support
- 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_TCDDataPublisher.cpp	- Actuators data publisher
mount_TCDDataSubscriber.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	- <i>test program</i>
sacpp_mount_pub	- <i>test program</i>
src	

mount_TC/cpp/standalone/src:

e.g. salgenerator mount sal java

java -

mount/java:

classes	- compiled type classes
mount	- generated java types
Makefile.saj_mount_types	- makefile for 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

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	- <i>run test programs</i>
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

Once the java has been generated it is also possible to create a Maven project for ease of distribution. Use the command e.g.

salgenerator mount maven

will create and build a maven project and save it in

\$SAL_WORK_DIR/maven/mount_[sal-version-number]

e.g. salgenerator mount sal python

mount/cpp/src :

Makefile_sacpp_mount_python	
SALPY_mount.cpp	- Boost.python wrapper
SALPY_mount.so	- import'able python library

4.2 salgenerator Options

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

validate	- check the XML files, generate validated IDL
html	- generate web form interfaces and documentation
labview	- generate LabVIEW interface
sal [lang]	- generate SAL C++, Java, or Python wrappers
lib	- generate the SAL shared library for a subsystem
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 API examples

The SAL code generation process also generates a comprehensive set of test programs so that correct operation of the interfaces can be verified.

Sample code is generated for the C++, Java, and Python target languages currently.

The sample code provides a simple command line test for

- publishing and subscription for each defined Telemetry type

- issuing and receiving each defined Command type

- generating and logging for each defined Event type.

In addition , GUI interfaces are provided to simplify the launching of Command and Event tests.

The procedure for generating test VI's for the LabVIEW interface is detailed in Appendix X. At present this is an interactive process, involving lots of LabVIEW dialogs.

5. Testing

5.1 Environment

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

```
idlpp
```

should produce

```
Usage: idlpp [-c preprocessor-path] [-b ORB-template-path]
[-n <include-suffix>] [-I path] [-D macro[=definition]] [-S] [-C]
[-l (c | c++ \ cpp \ isocpp \ cs \ java)] [-j [old]:<new>] [-d directory] [-i]
[-P dll_macro_name[,<h-file>]] [-o (dds-types | custom-psm | no-equality)] <filename>
```

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

```
salgenerator
```

should produce

```
SAL generator tool - Usage :

salgenerator subsystem flag(s)

where flag(s) may be

    validate - check the XML Telemetry/Command/LogEvent definitions
    sal      - generate SAL wrappers for cpp, java, isocpp, python
    lib      - generate shared library
    tcl      - generate tcl interface
    html     - generate web form interfaces
    labview  - generate LabVIEW low-level interface
    maven    - generate a maven repository
    db       - generate telemetry database table

    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

    sim      - generate simulation configuration
    icd      - generate ICD document
    link     - link a SAL program
```

verbose - be more verbose ; -)

Verify that the network interface is configured and operating correctly.

Make sure that `Firewalld` is properly configured (or disabled by issuing the `systemctl stop firewalld` command as root).

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_pub` - publisher
`skycam_<telemetryType>/cpp/standalone/sacpp_skycam_sub` - subscriber

start the subscriber first, then the publisher.

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_[command]_commander` - to send commands
`mount/cpp/src/sacpp_mount_[command]_controller` - to process commands

start the controller first, wait for it to print Ready, then run the commander

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

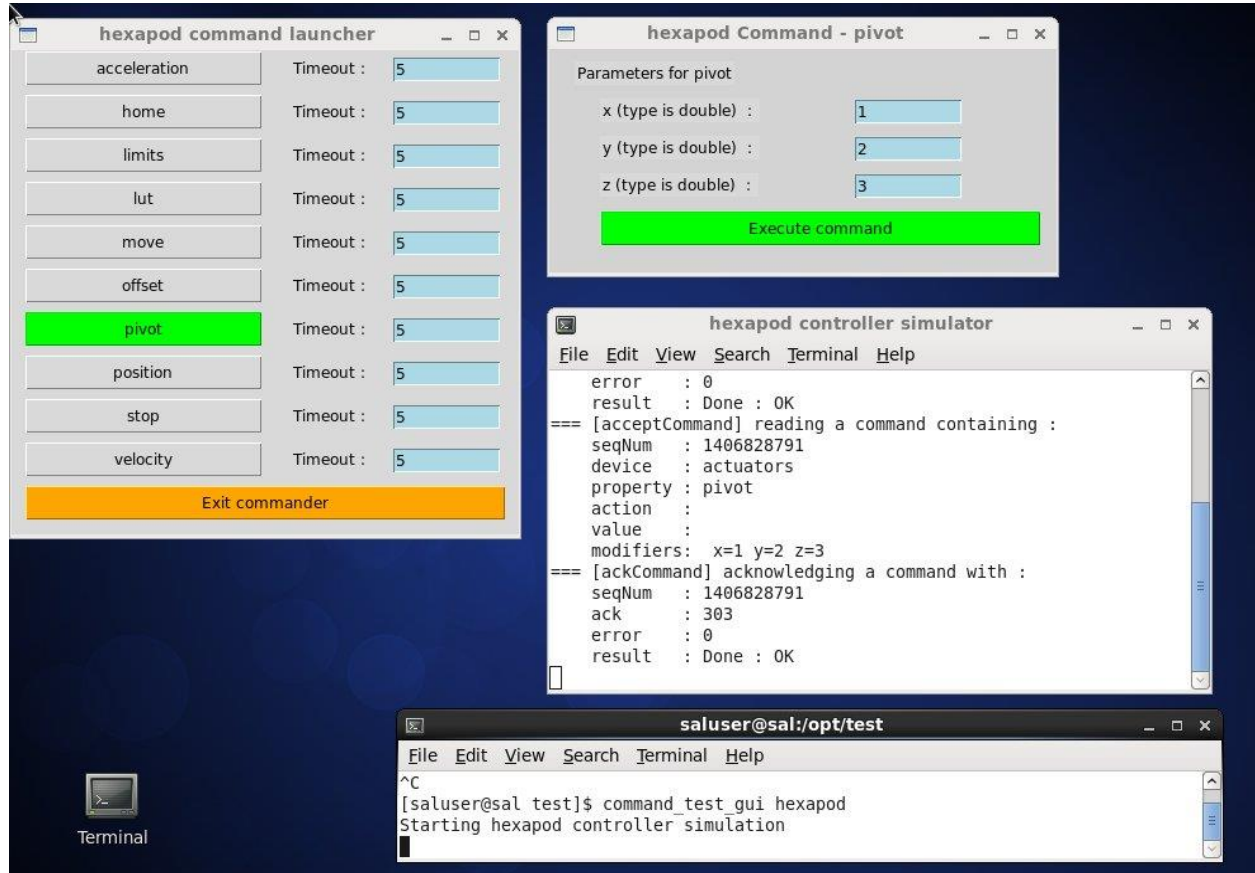
For C++

```
command_test_gui hexapod
```

This script is on the PATH, so you should be able to run it from the command line.

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 shows 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]_send - to generate events
mount/cpp/src/sacpp_mount_[event]_log - to log the events

start the event logger first and then the send.

For java

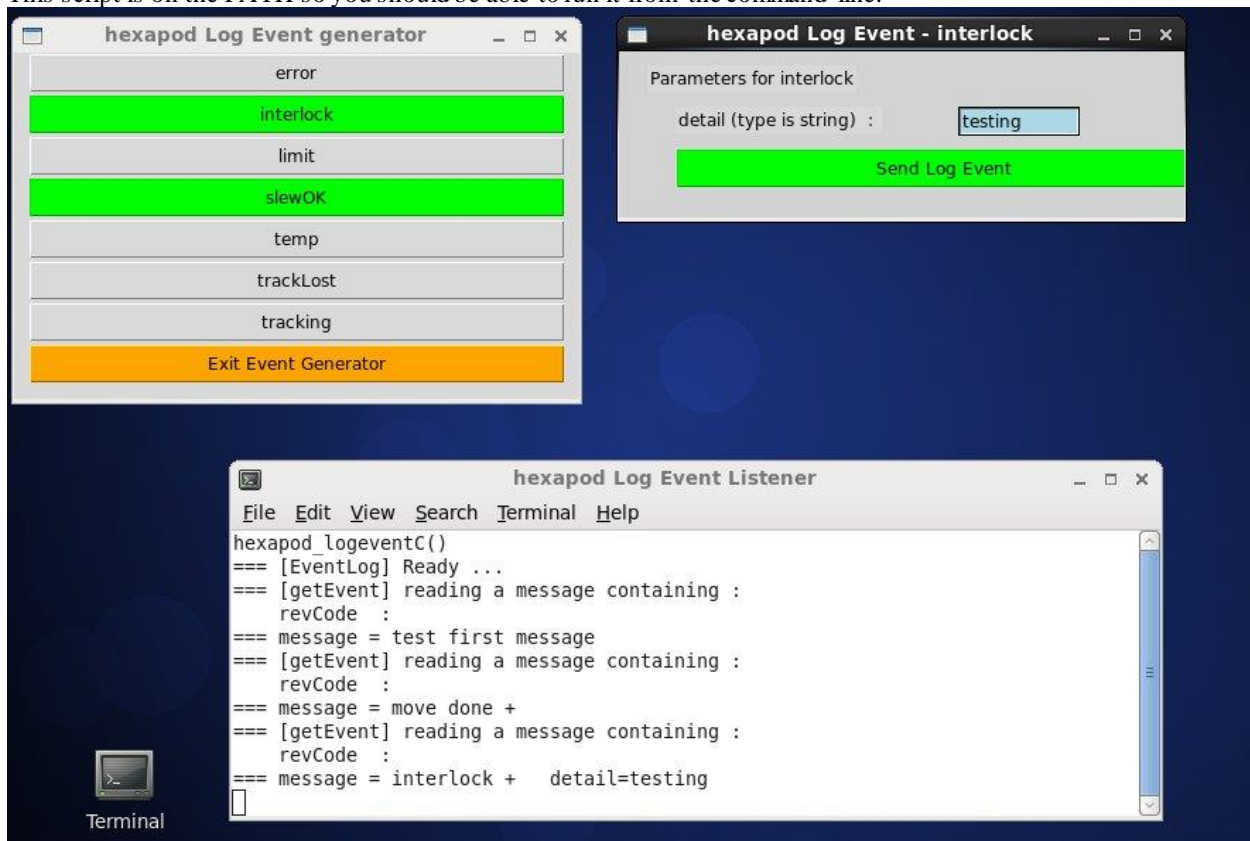
mount/java/src/mount_events.run - starts events processor

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

This script is on the PATH so you should be able to run it from the command line.



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.

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 <subsystem>; // substitute the actual subsystemname 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 publisher object
void salTelemetrySub(char *topicName);	- create telemetry subscriber object
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, long error, char *result);	- acknowledge a command
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 subsystemname here
import org.lst.sal.<SAL_subsystem>; //substitute actual subsystemname 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

<code>public int getProperty(String property, String value)</code>	- get configuration item
<code>public int setProperty(String property, String value)</code>	- set configuration item
<code>public void salCommand()</code>	- create a command object
<code>public void salProcessor()</code>	- create command processor object
<code>public void salShutdown()</code>	- tidyup
<code>public void salEvent(String topicName)</code>	- create event object

6.3 Python (pybind11 bindings)

Each Telemetry/Command/Event datatype is wrapped like this (arrays are mapped to numpy arrays).

```
py::class_<atcs_command_OffsetC>(m, "atcs_command_OffsetC")
    .def(py::init<>())
    .def_readwrite("device", &atcs_command_OffsetC::device)
    .def_readwrite("property", &atcs_command_OffsetC::property)
    .def_readwrite("action", &atcs_command_OffsetC::action)
    .def_readwrite("value", &atcs_command_OffsetC::value)
    .def_readwrite("offsetX", &atcs_command_OffsetC::offsetX)
    .def_readwrite("offsetY", &atcs_command_OffsetC::offsetY)
    ;
```

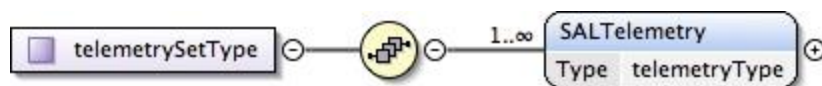
Commands , Events, and Telemetry calls are wrapped like this, every C++ method has a corresponding python binding.

```
.def( "issueCommand_enable",          &SAL_atcs::issueCommand_enable )
.def( "acceptCommand_enable",        &SAL_atcs::acceptCommand_enable )
.def( "ackCommand_enable",           &SAL_atcs::ackCommand_enable )
.def( "waitForCompletion_enable",     &SAL_atcs::waitForCompletion_enable
)
)
```

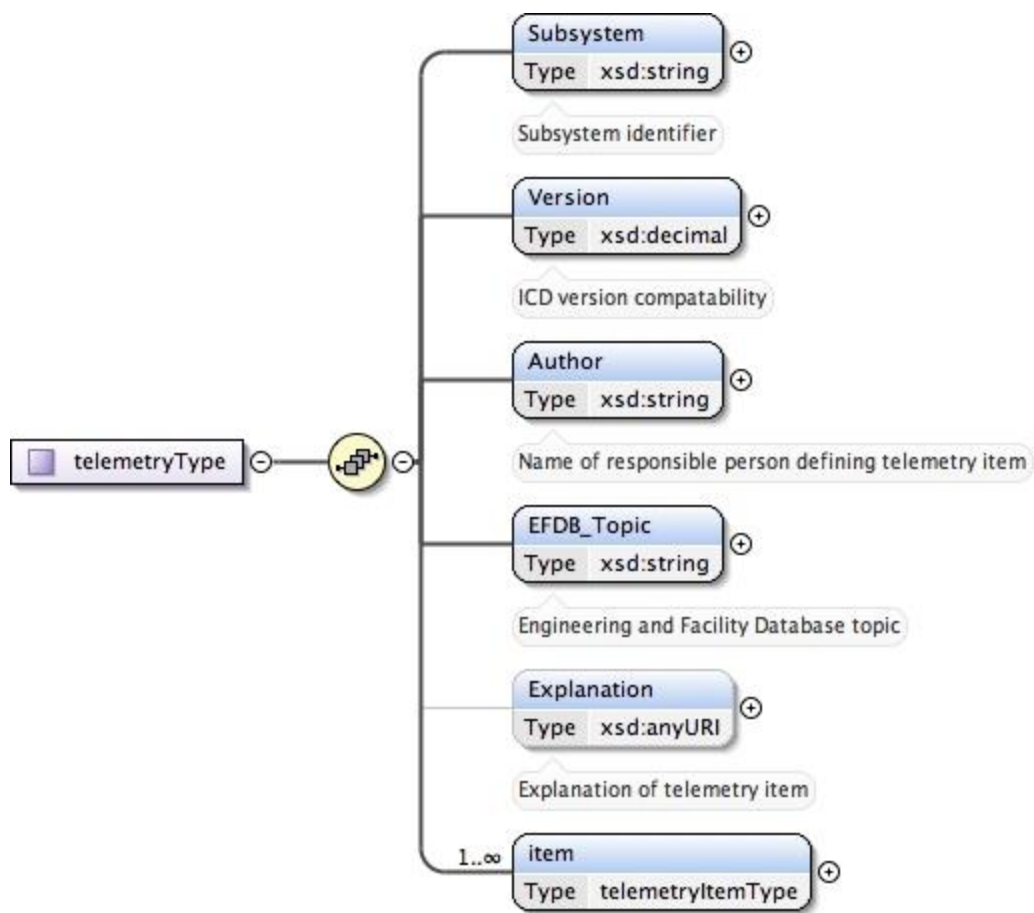
7.0 SAL XML Schema

7.1 Telemetry

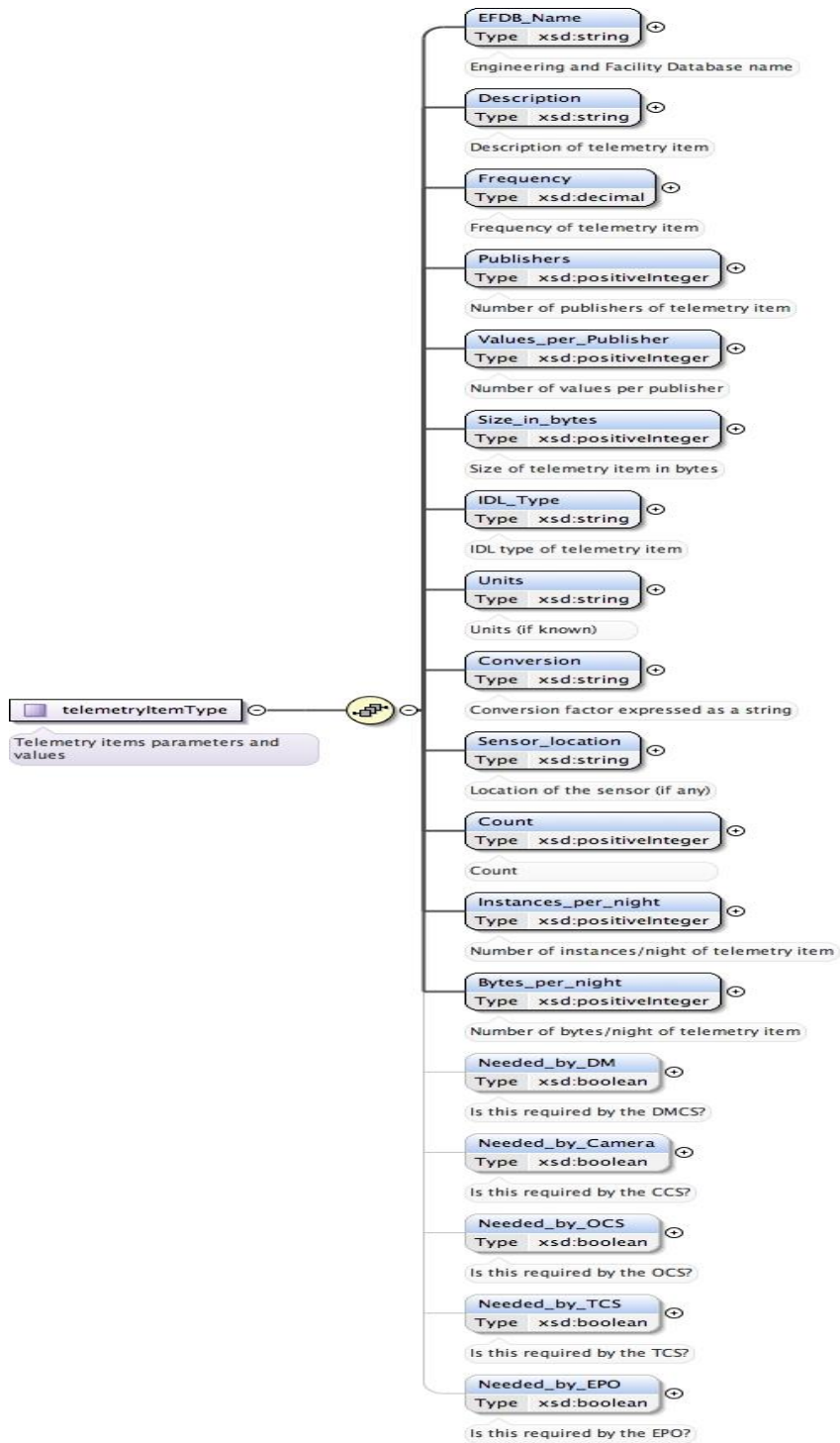
7.1.1 telemetrySetType



7.1.2 telemetryType



7.1.3 telemetryItemType

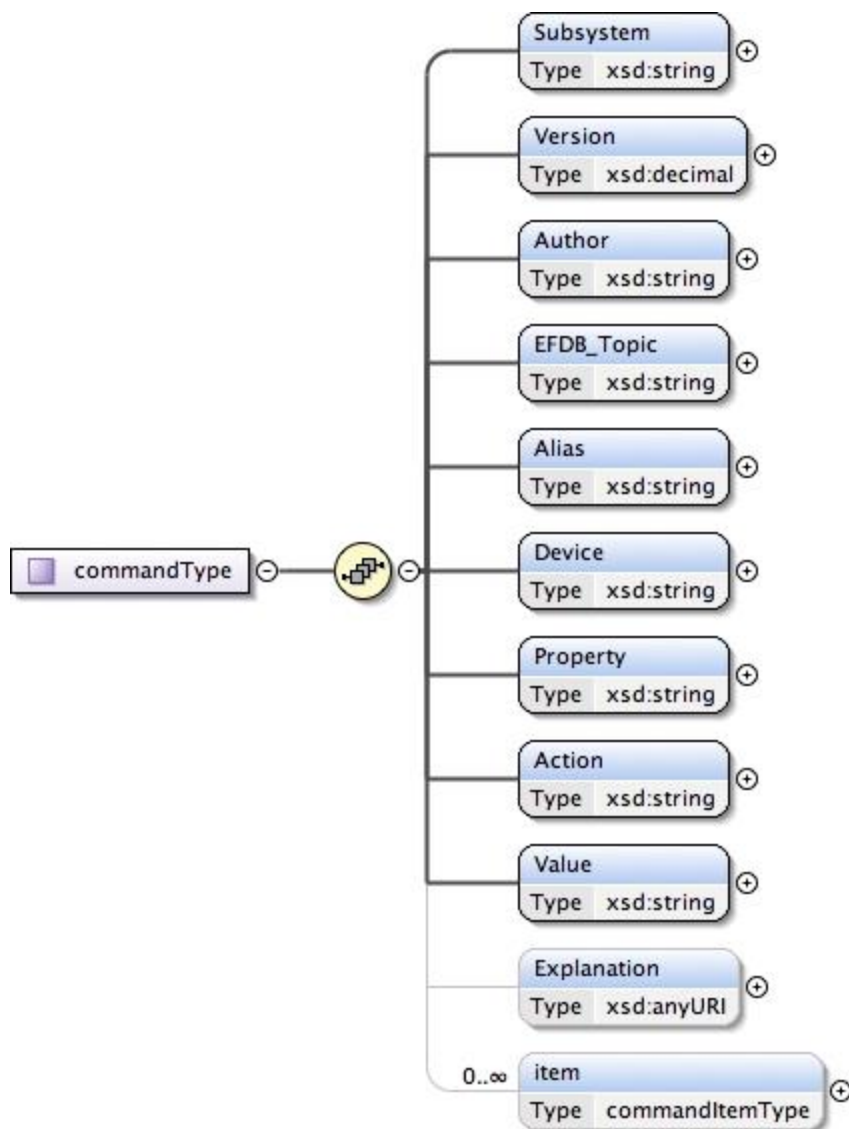


7.2 Commands

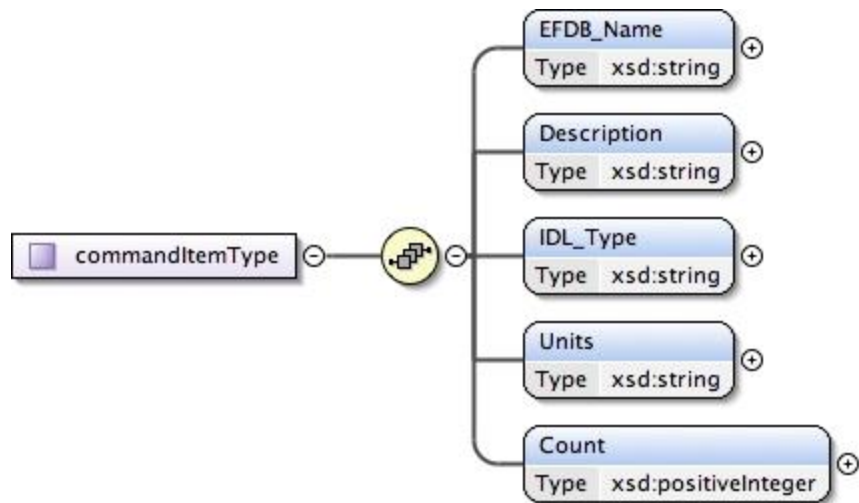
7.2.1 commandSetType



7.2.2 commandType



7.2.3 commandItemType

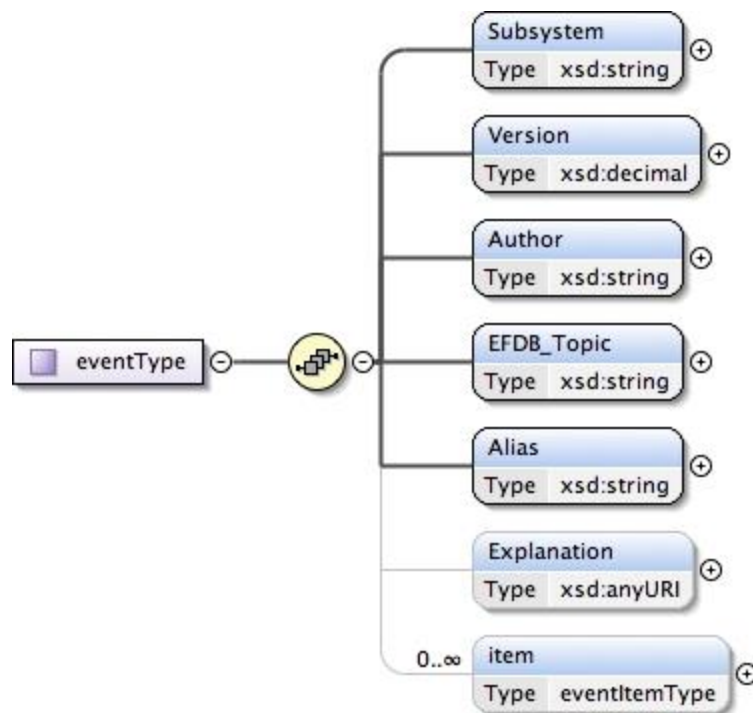


7.3 Events

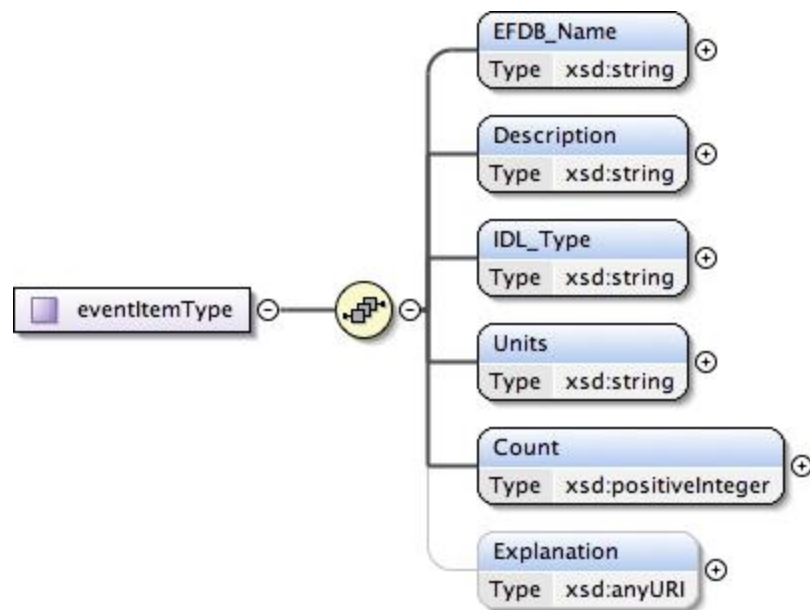
7.3.1 eventSetType



7.3.2 eventType



7.3.3 eventItemType



8.0 Compiler Options and Link Libraries

The following compiler options are required when compiling application code For subsequent linking with the SAL and DDS middleware.

CFLAGS/CXXFLAGS : -m64 -D_REENTRANT -fPIC -Wno-write-strings

Subsystems with duplicate instantiations (e.g. Hexapods) also require

-DSAL_SUBSYSTEM_IS_KEYED

and the following include paths will be required

-I\$(OSPL_HOME)/include
-I\$(OSPL_HOME)/include/sys
-I\$(OSPL_HOME)/include/dcps/C++/SACPP
-I\$(SAL_HOME)/include
-I\$(SAL_WORK_DIR)/include
-I../-subsys-/cppsrc

Where -subsys- is the subsystem name e.g. hexapod

The following libraries are required when linking an application to use the SAL and DDS middleware. For an application that communicates with multiple subsystems, the SAL libraries for each must be included.

SAL : libSAL_[subsystem-name].so , libsacpp_[subsystem-name]_types.so

DDS : libdcpsacpp.so , libdcpsgapi.so , libddsuser.so , libddskernel.so ,
libddsserialization.so , libddsconfparser.so , libddsdatabase.so , libddsutil.so,
libddsos.so, libddsconf.so

Other : libdl.so , libpthread.so

Appropriate linker path directives are

-L\$(OSPL_HOME)/lib -L\$(SAL_HOME)/lib

9.0 LabVIEW test VI generation

If you have multiple LabVIEW versions installed, or if LabVIEW is installed in a non default location, you can use the environment variable LABVIEW_HOME to control where the SDK looks for the LabVIEW header files.

e.g. `export LABVIEW_HOME=/opt/natinst`

would expect to find headers in `/opt/natinst/LabVIEW_20[xx]_64`

Run the salgenerator steps in order

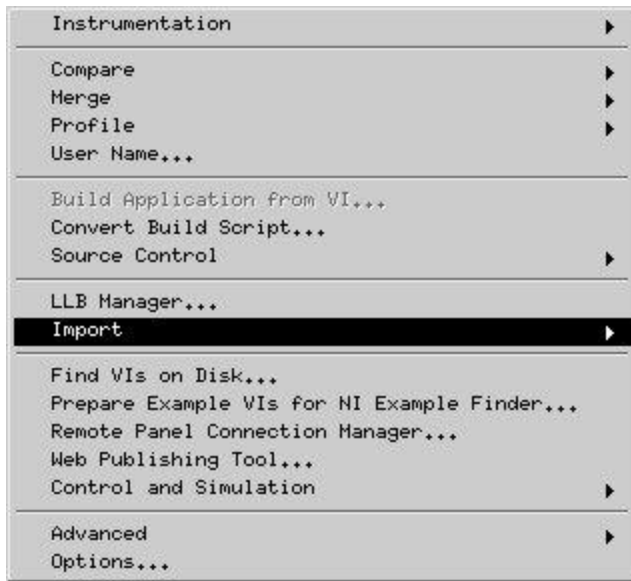
```
salgenerator [subsystem] validate
salgenerator [subsystem] sal cpp
salgenerator [subsystem] labview
```

The generation of the LabVIEW test VI's is an interactive process. The LabVIEW Shared library import is used to automatically generate VI's to interact with the Salgenerator produced SALLV_[subsystem].so library.

NOTE : It is vital to COMPLETELY DELETE the entire destination directory and it's Contents so that wizard can create it's output directory afresh. For example if you choose To place the results in `/home/me/sal/test/tcs/labview/lib`, then you should run the following command BEFORE starting the LabVIEW tools.

```
rm -fr /home/me/sal/test/tcs/labview/lib
```

1. Start LabVIEW and select the Tools->Import->Shared Library (.so) option



2. Choose either New or Update option and specify the path to the library and then click Next. Proceed through the rest of the dialogs as illustrated below. Generally selecting the default and clicking Next is appropriate.

The only non-standard option is in the “Configure Include Paths...” dialog where you must enter the


`BUILD_FOR_LV=1`

Option in the Preprocessor options section.

[illegible]


Import Shared Library

Select Shared Library and Header File



Shared Library (.so) File


/home/dmills/sal/test/m2ms/labview/SALLV_m2ms.so



☐ Shared library file is not on the local machine

Header (.h) File

/home/dmills/sal/test/m2ms/labview/SAL_m2ms_shmem.h



Back

Next

Cancel

Help

Import Shared Library

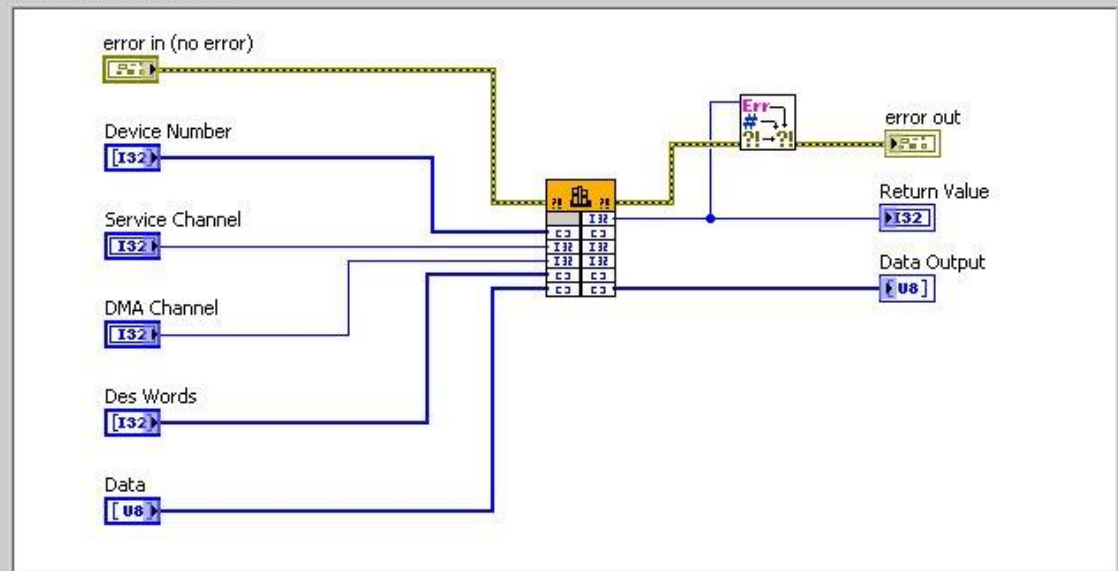
Select Error Handling Mode



Error Handling Mode

Function Returns Error Code/Status

Example Block Diagram



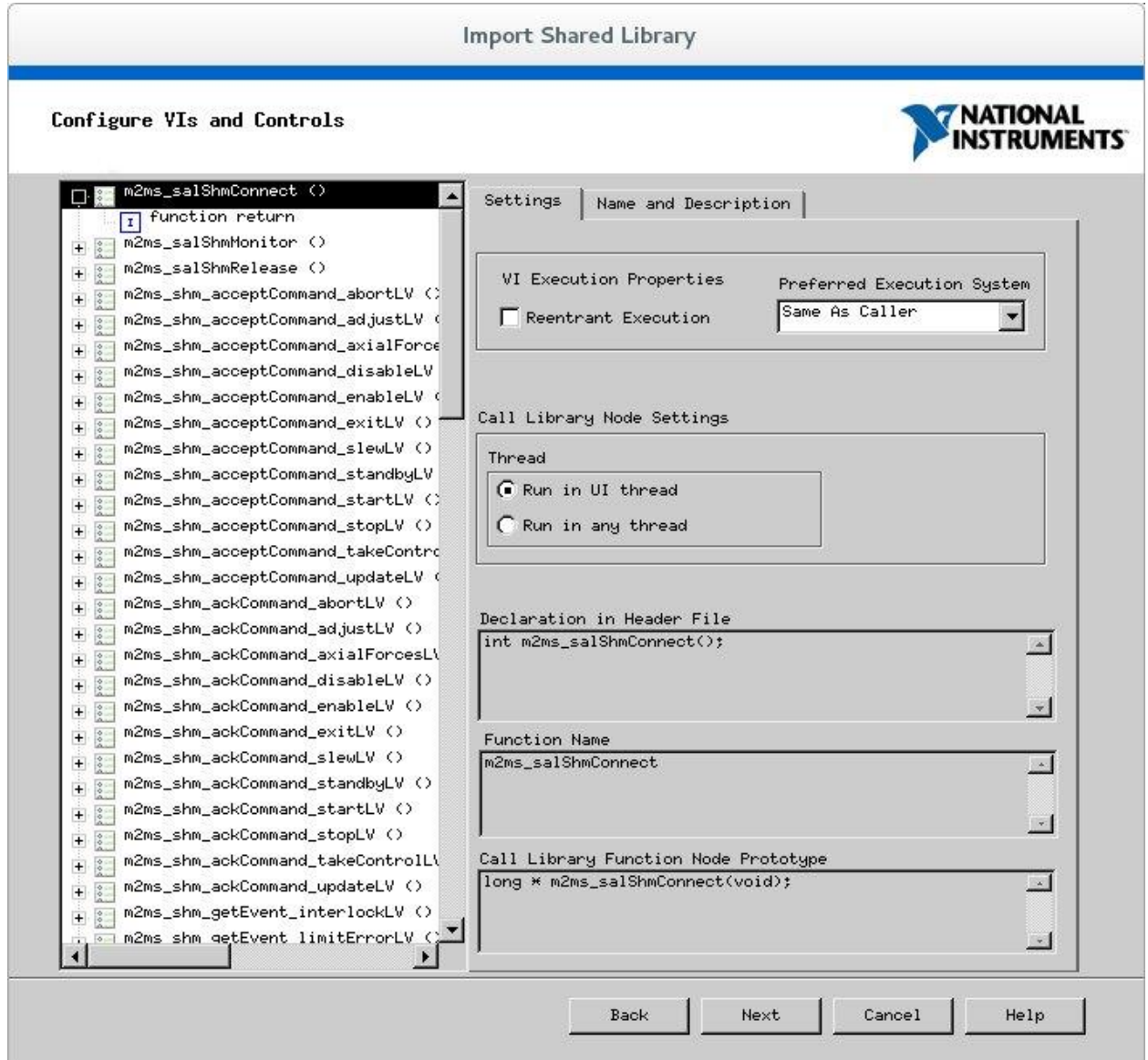
You want to call the generated function only when there are no errors in.
The function returns an error code/status and converts that error or warning code to an error cluster.

Back

Next

Cancel

Help



Import Shared Library

Generation Summary



```

The selected shared library and head file:
/
/home/dmills/sal/test/m2ms/labview/SAL_m2ms_shmem.h

The generated files are installed in the following folder:
/home/dmills/sal/test/m2ms/labview/lib

The generated lvlib name:
SALLV_m2ms.lvlib

The error handling mode:
Function Returns Error Code/Status

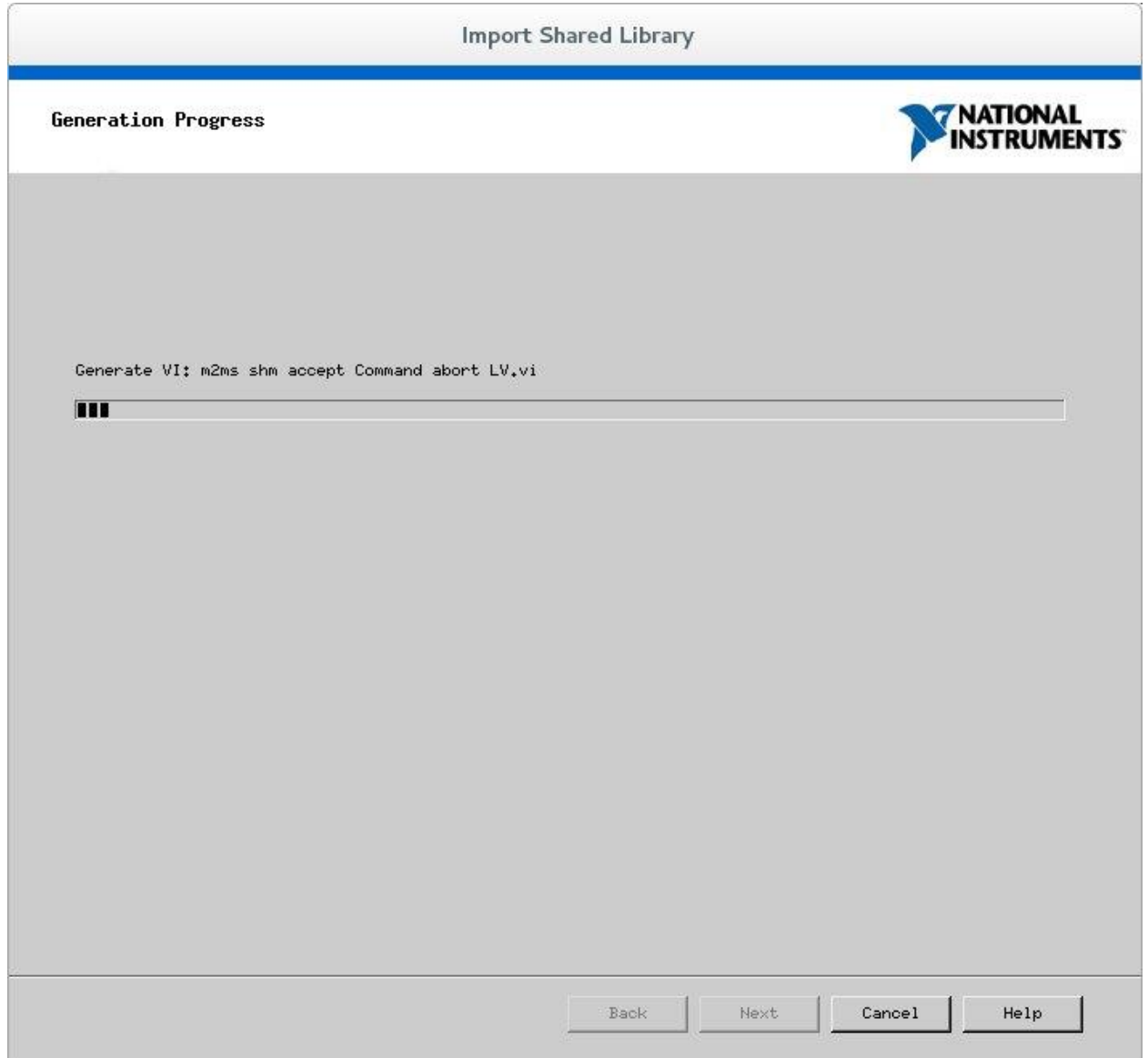
Total number of selected function(s): 96
int m2ms_salShmConnect();
int m2ms_salShmMonitor();
int m2ms_salShmRelease();
int m2ms_shm_acceptCommand_abortLV(m2ms_command_abortC **command_abort_Ctl);
int m2ms_shm_acceptCommand_adjustLV(m2ms_command_adjustC **command_adjust_Ctl);
int m2ms_shm_acceptCommand_axialForcesLV(m2ms_command_axialForcesC **command_axialForces_Ctl);
int m2ms_shm_acceptCommand_disableLV(m2ms_command_disableC **command_disable_Ctl);
int m2ms_shm_acceptCommand_enableLV(m2ms_command_enableC **command_enable_Ctl);
int m2ms_shm_acceptCommand_exitLV(m2ms_command_exitC **command_exit_Ctl);
int m2ms_shm_acceptCommand_slewLV(m2ms_command_slewC **command_slew_Ctl);
int m2ms_shm_acceptCommand_standbyLV(m2ms_command_standbyC **command_standby_Ctl);
int m2ms_shm_acceptCommand_startLV(m2ms_command_startC **command_start_Ctl);
int m2ms_shm_acceptCommand_stopLV(m2ms_command_stopC **command_stop_Ctl);
int m2ms_shm_acceptCommand_takeControlLV(m2ms_command_takeControlC **command_takeControl_Ctl);
int m2ms_shm_acceptCommand_updateLV(m2ms_command_updateC **command_update_Ctl);
int m2ms_shm_ackCommand_abortLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_adjustLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_axialForcesLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_disableLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_enableLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_exitLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_slewLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_standbyLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_startLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_stopLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_takeControlLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_ackCommand_updateLV(int cmdSeqNum, salLONG ack, salLONG error, char *result);
int m2ms_shm_getEvent_interlockLV(m2ms_logevent_interlockC **pdata);
int m2ms_shm_getEvent_limitErrorLV(m2ms_logevent_limitErrorC **pdata);
int m2ms_shm_getEvent_tempErrorLV(m2ms_logevent_tempErrorC **pdata);
int m2ms_shm_getNextSample_ackcmdLV(m2ms_ackcmdC **ackcmd_Ctl);
int m2ms_shm_getNextSample_axial_actuatorsLV(m2ms_axial_actuatorsC **axial_actuators_Ctl);
int m2ms_shm_getNextSample_ni9201LV(m2ms_ni9201C **ni9201_Ctl);
    
```

Back

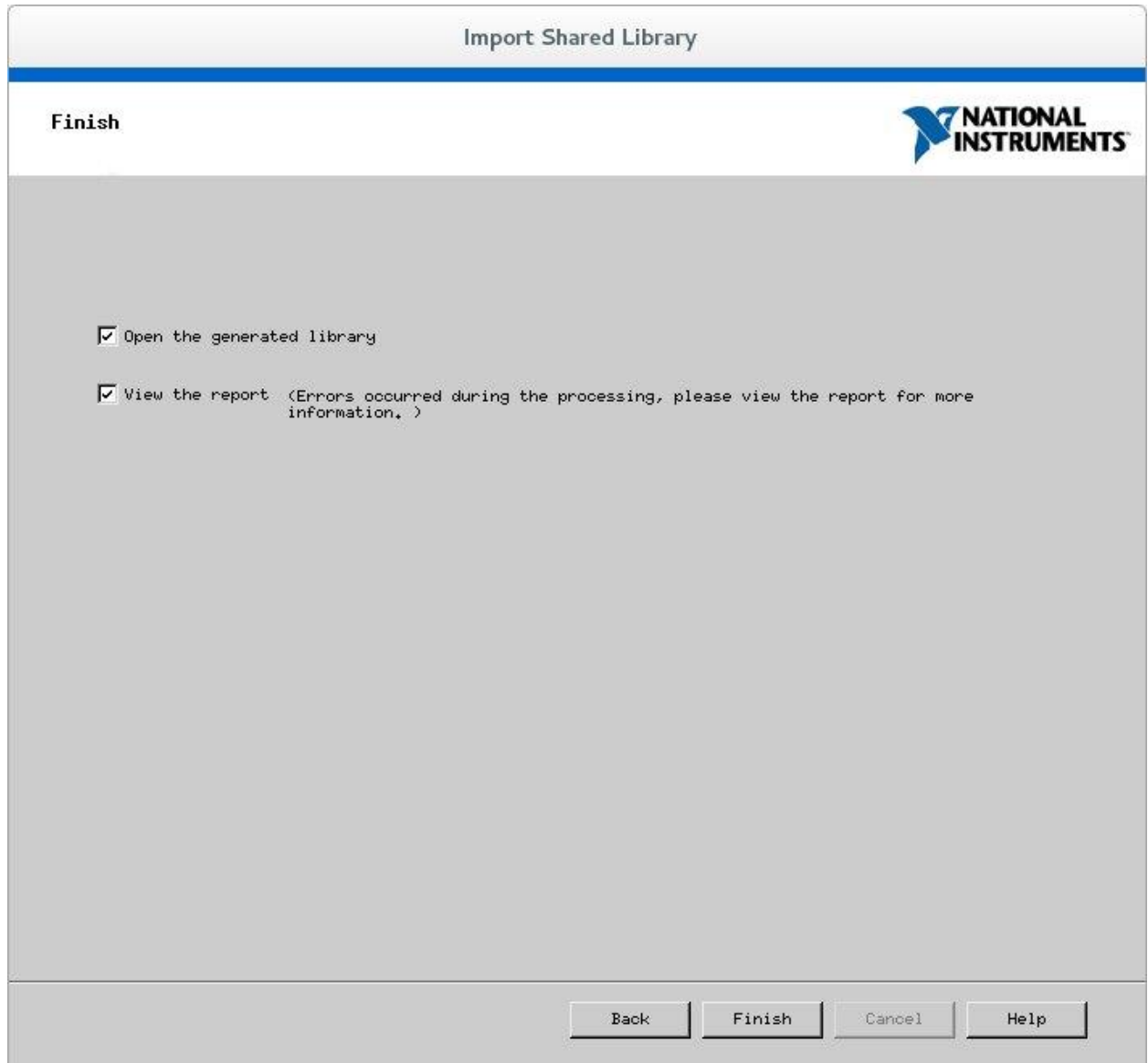
Next

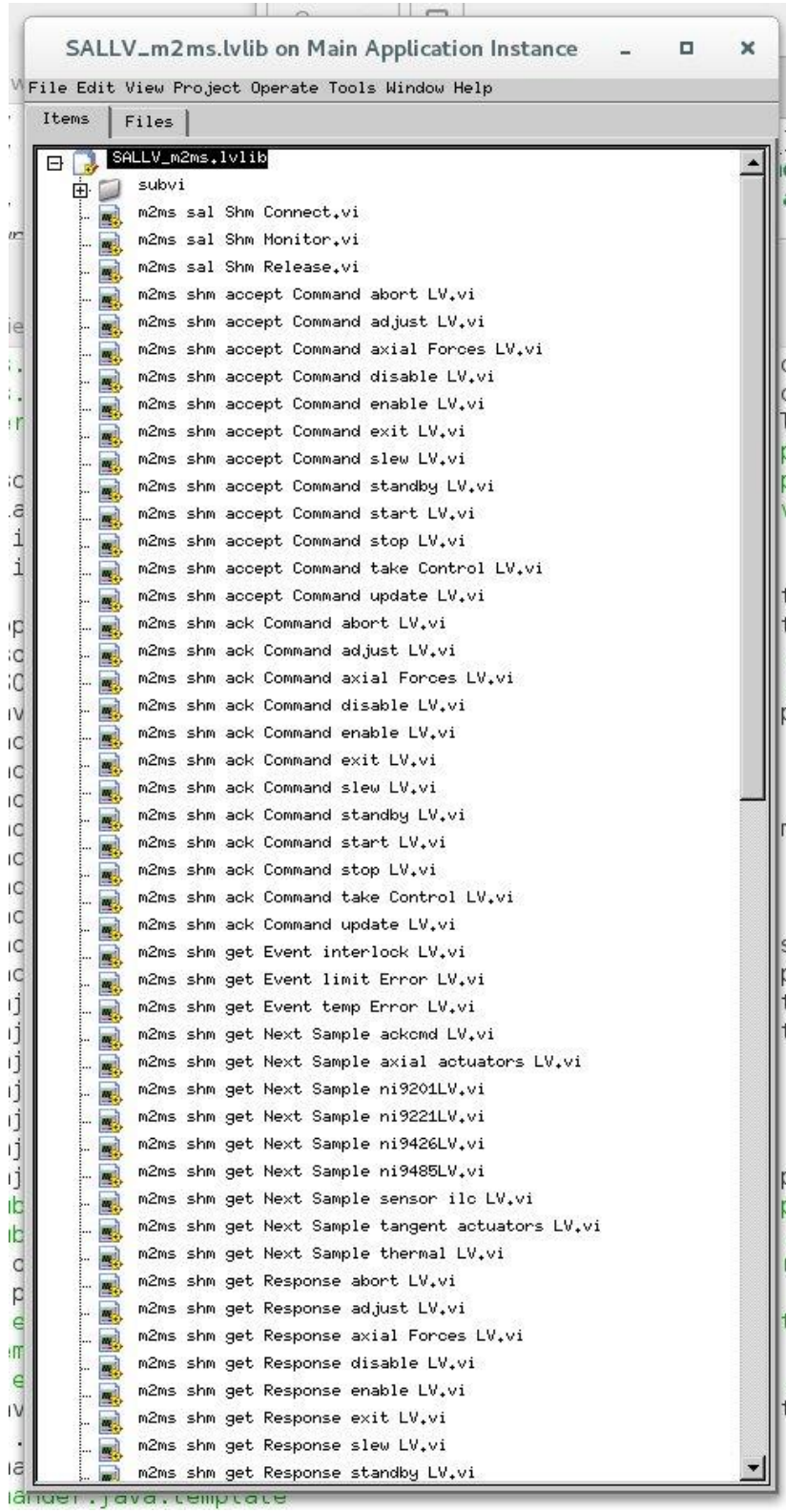
Cancel

Help



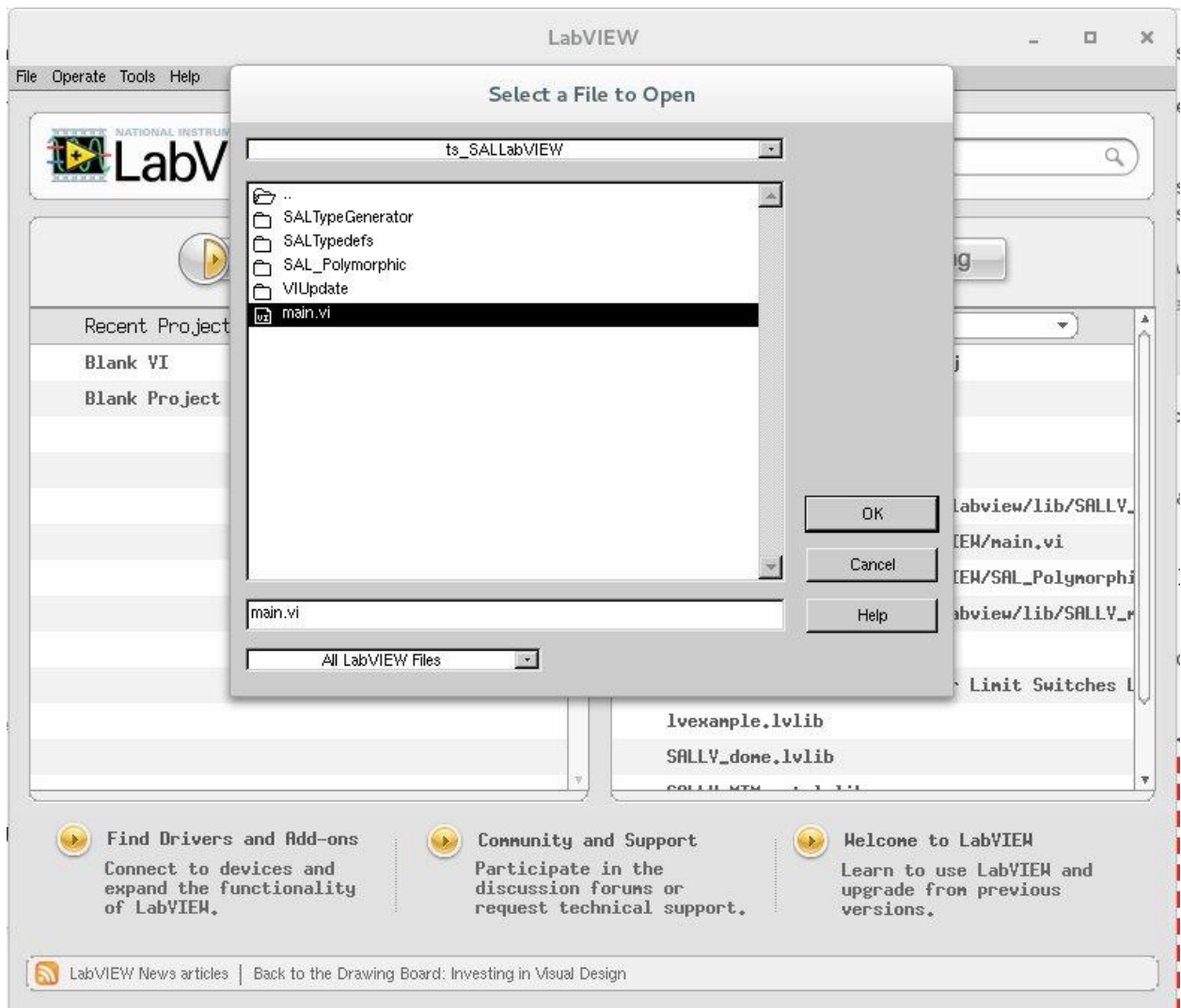
Click **Finish** on the dialog.



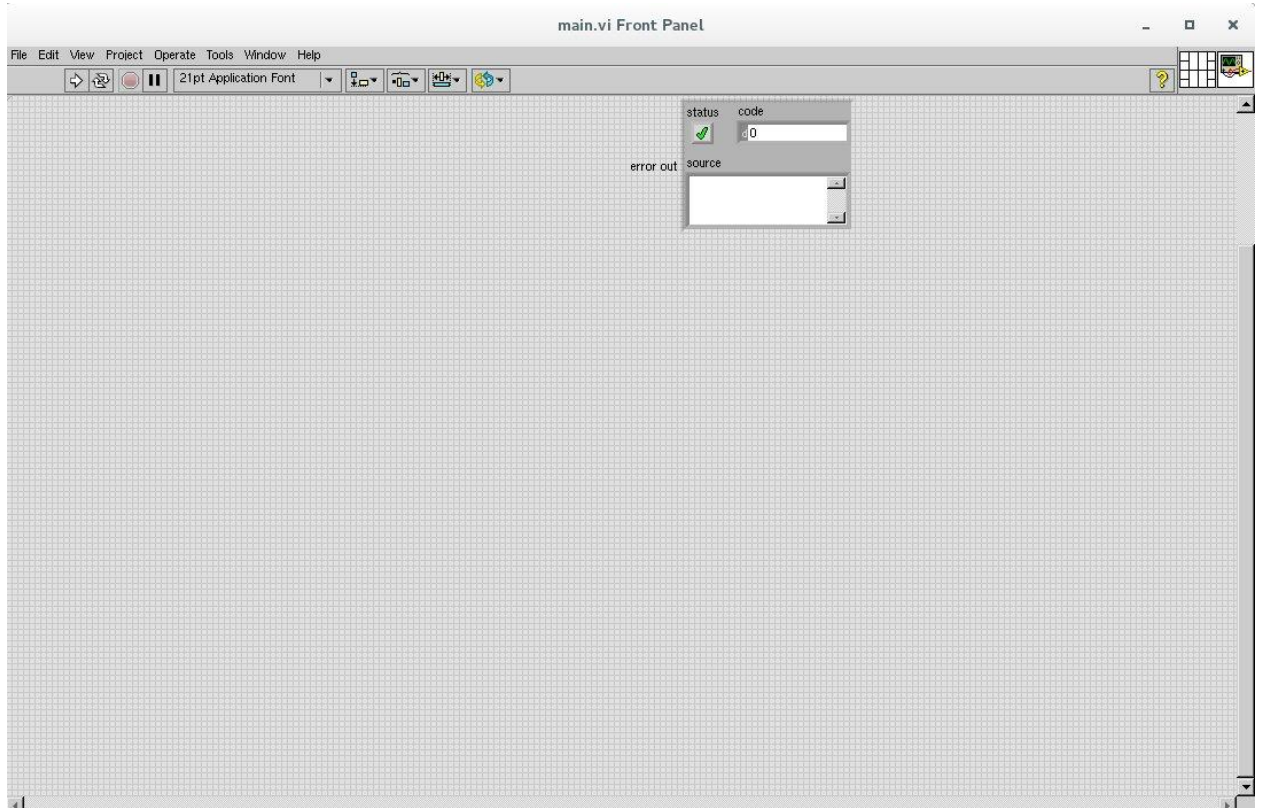


When the LabVIEW import library wizard has completed it is necessary to run another LSST provided VI to finish the generation process.

Use the LabVIEW File->Open dialog to locate ts_SALLabVIEW/main.vi



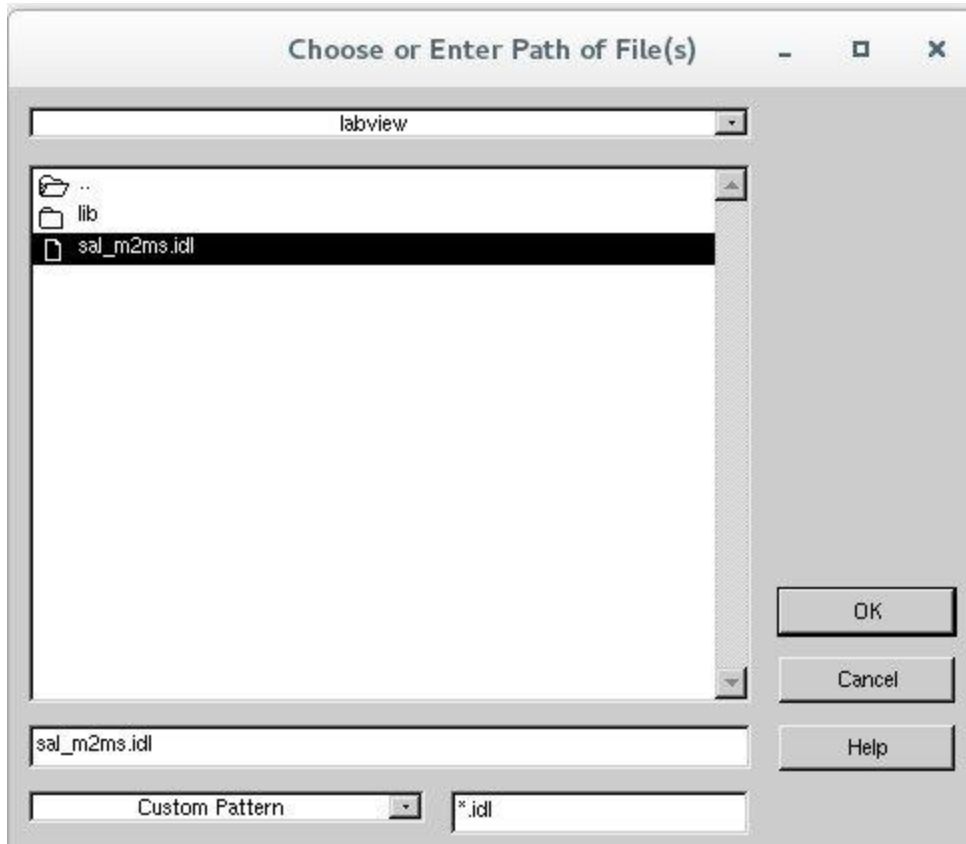
Click OK to run the main.vi VI. It will open a mostly empty interface.



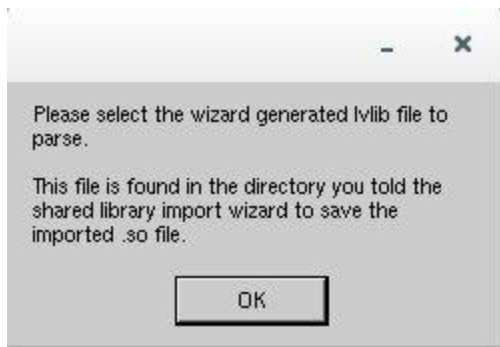
Click the Run icon.



Click OK and select the subsystem IDL file. The correct file should be found in the [subsystem]/labview directory of the SAL_WORK_DIR tree.

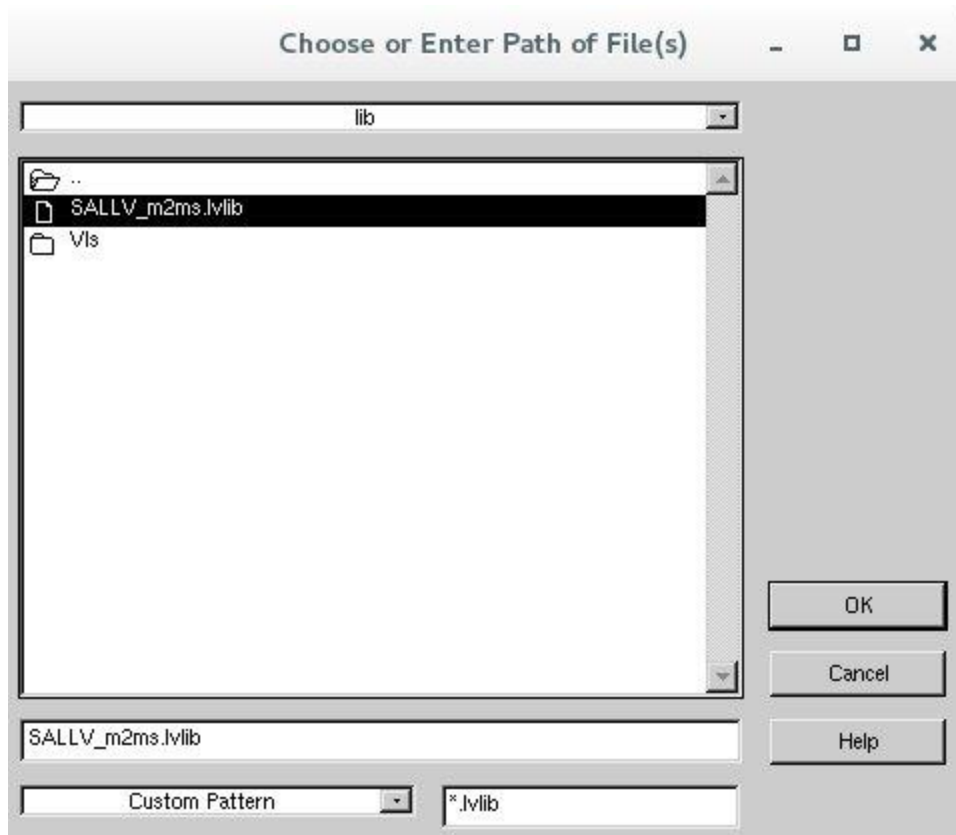


Click OK to select it.



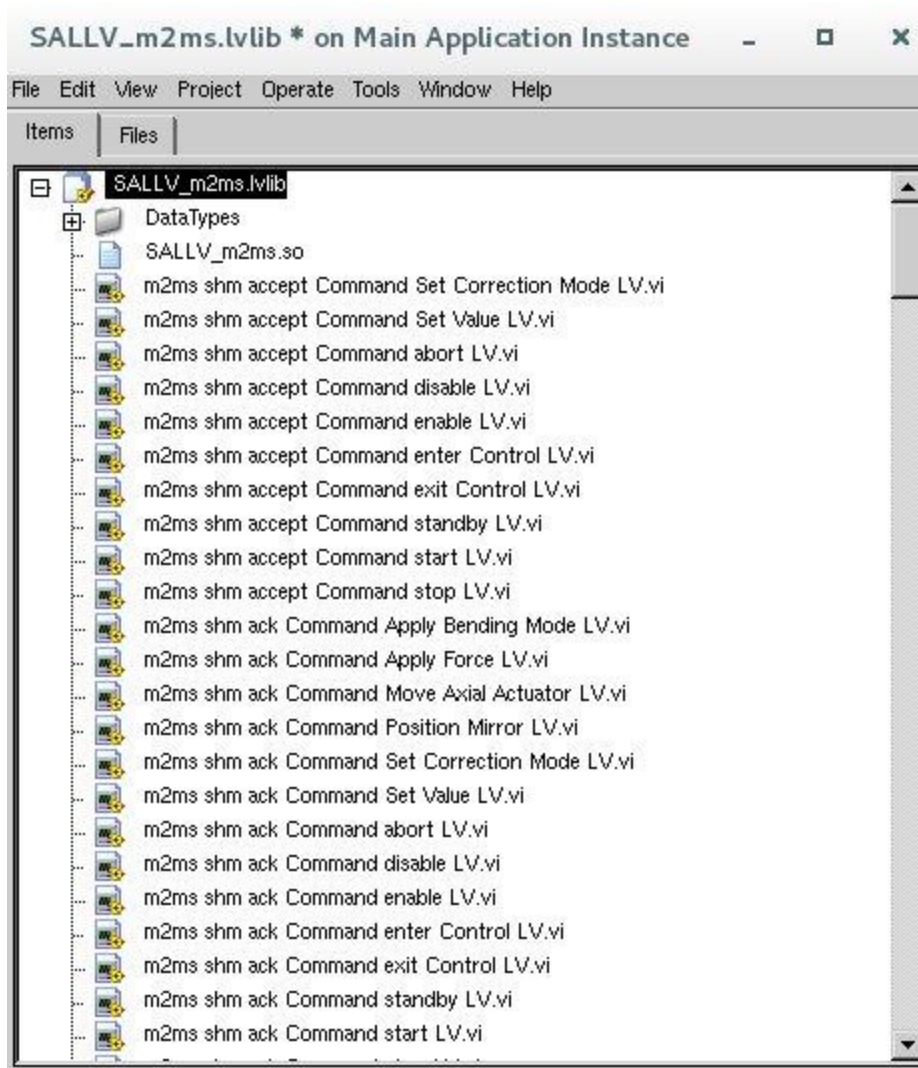
Click OK

Another file dialog then appears for you to select the .lvlib containing the VI's. This should be located in the [subsystem]/labview/lib directory of the SAL_WORK_DIR tree.



Click OK.

There will then be an extensive period where multiple windows flash on the screen as each VI is individually processed. Finally a library contents window will appear.



Another extensive period will follow where each VI is processed again (you will see them being removed and re-added to the list one-by-one).

Finally the process completes and the main LabVIEW window will reappear.

Once the VI's has been built, you can manually test them by running them against either each other , or against the C++/Java/Python test programs.

Regardless of which option you choose, the LabVIEW environment must be set up first by

1. Running the SALLV_[subsystem]_Monitor daemon in a terminal (this executable manages the shared memory used to mediate the transfer of data to and from LabVIEW). The daemon will have been built in the [SAL_WORK_DIR]/[subsystem]/labview directory.
2. Run the [subsystem]_shm_connect VI and leave it open
3. Depending upon the required function, an initialization VI should be run i.e. for command receivers , run [subsystem]_shm_salProcessor_[name], for event receivers , run [subsystem]_shm_salEvent_[name], and for Telemetry receivers , run [subsystem]_shm_salTelemetrySub.
4. After an application has completed all it's SAL mediated communications, it is essential to call the [subsystem]_shm_release VI to clean up.