SLS Detector Documentation

Release trunk

A. Parenti

Jul 15, 2025

Contents

Contents:

CHAPTER 1

Introduction

The slsDetectors package allows you to control a Gotthard, or another Detector supported by PSI's slsDetectorPackage. The documentation from PSI can be found here.

CHAPTER 2

Expert Contact

- Andrea Parenti <andrea.parenti@xfel.eu>

CHAPTER 3

Deployment Guidelines

slsDetectors will automatically install its dependency *slsDetectorsPackage*.

For debugging purposes it can be useful to have `tshark` installed on the control server, and `xctrl` user added to the `wireshark` group.

The control server should have a 10 GbE network interface for sending images to the DAQ, and possibly one for the GUI server.

3.1 Special settings for Jungfrau

In order to have the receiving thread executed with real time priority, the line

```
username rtprio 99
```

shall be added to the file `/etc/security/limits.conf` on the control server (may differ depending on the Linux distribution).

The RX socket buffer size shall be set to 1000 MB

```
sysctl -w net.core.rmem_max=1048576000
```

The maximum socket input packet queue shall be set to 250000

```
sysctl -w net.core.netdev_max_backlog=250000
```

The MTU on the network interface used for Jungfrau shall be set to 9000, also known as “jumbo frames”.

CHAPTER 4

Receiver Device Setup

The following parameter is needed if multiple receivers are run on the same control host:

- *rxTcpPort*: the port on which the receiver will wait for reconfiguration from the control device. It corresponds to the *–rx_tcpport* or *-t* option in the *slsReceiver* from PSI. It must be unique on the control host.

CHAPTER 5

Control Device Setup

The following are mandatory parameters for the control device. As one single device can control several detectors, all the parameters are vectors. Also, in parenthesis the corresponding parameter name in PSI's tools is provided:

- *detectorHostName* (*hostname*): the hostnames or IP addresses of the detector modules to be controlled;
- *udpSrcIp* (*udp_srcip*): the IP addresses of the detector (source) UDP interfaces; must be in the same subnet as the destination UDP/IP; used to be named *detectorIp* (*detectorip*);
- *rxHostname* (*rx_hostname*): the hostnames or IP addresses of the hosts where the receiver devices are running;
- *rxTcpPort* (*rx_tcpport*): the port on which the receiver for the detector has been started; see the *rxTcpPort* paramter of the Karabo receiver.
- *udpDstIp* (*udp_dstip*): the IP addresses of the network interfaces on the receiver's host, which is receiving data from the detector; used to be named *rxUdpIp* (*rx_udpip*);
- *udpDstPort* (*udp_dstport*): the port numbers of the receiver (destination) UDP interfaces; default is 50001; it must be unique if multiple receivers are run on the same host; used to be named *rxUdpPort* (*rx_udpport*).

CHAPTER 6

How to control an SLS detector

The SlsControl class allows you to control a Gotthard, or another Detector supported by PSI's slsDetectorPackage. You can set parameters, start and stop an acquisition.

The data receiver must be also running. It can be for example the slsReceiver provided with the slsDetectorPackage, or a SlsReceiver Karabo device.

6.1 Mandatory parameters

The mandatory parameters are described in *Control Device Setup*.

6.2 Optional parameters

The SlsControl device has several optional parameters; please be aware that:

- simple (i.e. *scalar*) reconfigurable parameters tagged as *sls* are sent to all the modules;
- vector reconfigurable parameters tagged as *sls* must have zero, one, or as many elements as the number of modules. If the vector has only one element, this value will be sent to all modules;

CHAPTER 7

How to create a control device based on slsControl

If your detector is not covered by an already existing SlsControl derived class, it can easily added to the slsControl package. You have to create the source and header files (see *MyControl.hh file* and *MyControl.cc file* Sections) and add them to the Netbeans project. You also have to compile the project in Netbeans (Debug, Release and Simulation mode) in order to have the Makefiles updated.

Do not forget to "git add" the new files.

7.1 Expected parameters

There are already several common detector parameters defined in the base class (SlsControl). Check the SlsControl::expectedParameter() function to see what they are.

In case you need to set more parameters, they must have the "sls" tag. The alias have to contain the parameter name, as can be found in the description of the command line interface.

Any Karabo type is allowed, also VECTOR types.

7.2 MyControl.hh file

This is the minimal MyControl.hh:

```
#ifndef KARABO_MYCONTROL_HH
#define KARABO_MYCONTROL_HH


#include <karabo/karabo.hpp>

#include "SlsControl.hh"
#include "version.hh"  // provides PACKAGE_VERSION

/**
```

(continues on next page)

(continued from previous page)

```
 * The main Karabo namespace
 */
namespace karabo {

    class MyControl : public karabo::SlsControl {
    public:

        KARABO_CLASSINFO(MyControl, "MyControl", PACKAGE_VERSION)

        MyControl(const karabo::util::Hash& config);

        virtual ~MyControl();

        static void expectedParameters(karabo::util::Schema& expected);

    private:

        // Optional. Send the commands needed to power-up and initialize
        // the detector
        void powerOn();

        // Optional. Place here the code needed to regularly poll detector
        // parameters, e.g. temperatures, and set them in the input Hash.
        void pollDetectorSpecific(karabo::util::Hash& h);

        // Optional. Place here the code needed to execute detector specific
        // actions, when a reconfiguration is received.
        void configureDetectorSpecific(const karabo::util::Hash& configHash);

        // Optional. Place here the code needed to create the calibration
        // and settings files, if needed by the detector.
        void createCalibrationAndSettings(const std::string& settings);
    };

} /* namespace karabo */

#endif /* KARABO_MYCONTROL_HH */
```

You probably don't need anything more than that.

7.3 MyControl.cc file

An example of MyControl.cc is the following. In the best case you will just have to add detector specific expected parameters as described in the *Expected parameters* Section.

```
#include "MyControl.hh"

USING_KARABO_NAMESPACES

namespace karabo {

    KARABO_REGISTER_FOR_CONFIGURATION(BaseDevice, Device<>, SlsControl,
        MyControl)
```

(continues on next page)

(continued from previous page)

```
    MyControl::MyControl(const Hash& config) : SlsControl(config) {
    }

    MyControl::~MyControl() {
    }

    void MyControl::expectedParameters(Schema& expected) {
        // Add here more detector specific expected parameters, for
        // example:

        VECTOR_INT32_ELEMENT(expected).key("tempAdc")
            .displayedName("ADC Temperature")
            .unit(Unit::DEGREE_CELSIUS)
            .readOnly()
            .commit();
    }

    void MyControl::powerOn() {
        // Send the commands needed to power-up and initialize the
        // detector, for example:
        sendConfiguration("powerchip", "1");
    }

    void MyControl::pollDetectorSpecific(karabo::util::Hash& h) {
        // Poll detector specifica properties, for example temperatures:
        const std::vector<int> tempAdc = m_SLS->
↪getTemperature(slsDetectorDefs::dacIndex::TEMPERATURE_ADC, m_positions);
        h.set("tempAdc", tempAdc);
    }

    void MyControl::configureDetectorSpecific(const karabo::util::Hash& configHash) {
        // Execute detector specific actions, when a reconfiguration is
        // received.
    }


    void MyControl::createCalibrationAndSettings(const std::string& settings) {
        // Place here the code needed to create the calibration and
        // settings files, if needed by the detector. For example:

        const std::string calibrationDir = m_tmpDir + "/" + settings;

        if (!fs::exists(calibrationDir)) {
            // Create calibration and settings directory
            fs::create_directory(calibrationDir);

        KARABO_LOG_FRAMEWORK_DEBUG << "Created calibration dir" << calibrationDir;
        }

        const std::string fname = calibrationDir + "/calibration.sn";
        if (!fs::exists(fname)) {
            // Create calibration file
            std::ofstream fstr;
            fstr.open(fname.c_str());

            if (fstr.is_open()) {
               fstr << "227 5.6\n"
```

(continues on next page)

(continued from previous page)

```
                fstr.close();

            } else {
                throw KARABO_RECONFIGURE_EXCEPTION("Could not open file " + fname +
↪"for writing");
            }
        }
    }

} /* namespace karabo */
```

7.4 Simulation Mode

To compile slsControl in simulation mode, just run

```
make CONF=Simulation
```

This way the package will be linked against the simulation, instead of libSlsDetector.

For more details on how the simulation is implemented, see *The slsDetectorSimulation* Section.

CHAPTER 8

How to receive data from an SLS detector

The SlsReceiver class allows you to receive data from a Gotthard, or another Detector supported by PSI's slsDetector-Package.

The detector configuration, and the acquisition command, must be executed somewhere else, for example by the command line interface provided by PSI, or by a SlsControl Karabo device.

CHAPTER 9

How to create a receiver device based on slsReceiver

If your detector is not covered by an already existing SlsReceiver derived class, it can easily added to the slsReceiver package. You have to create the source and header files (see *MyReceiver.hh file* and *MyReceiver.cc file* Sections) and add them to the Netbeans project. You also have to compile the project in Netbeans (Debug, Release and Simulation mode) in order to have the Makefiles updated.

Do not forget to "git add" the new files.

9.1 Expected parameters

The receiver TCP port number (rx_tcpport) is already one of the expected parameters of the base class (SlsReceiver):

```
UINT16_ELEMENT(expected).key("rxTcpPort")
        .tags("sls")
        .alias("--rx_tcpport")
        .displayedName("rxTcpPort")
        .description("Receiver TCP Port")
        .assignmentOptional().defaultValue(1954)
        .init()
        .commit();
```

If you need to pass more options to the slsReceiverUsers object when it is instantiated in Karabo, you can do it by adding more expected parameters to your MyReceiver class. As done for the above one (rx_tcpport), you will need to tag them as "sls", and to give in their alias the parameter name.

9.2 MyReceiver.hh file

This is the minimal MyReceiver.hh:

```
#ifndef KARABO_MYRECEIVER_HH
#define KARABO_MYRECEIVER_HH

#include <karabo/karabo.hpp>

#include "SlsReceiver.hh"
#include "version.hh"  // provides PACKAGE_VERSION

/**
 * The main Karabo namespace
 */
namespace karabo {

    class MyReceiver : public karabo::SlsReceiver {

    public:

        KARABO_CLASSINFO(MyReceiver, "MyReceiver", "PACKAGE_VERSION")
        static void expectedParameters(karabo::util::Schema& expected);

        MyReceiver(const karabo::util::Hash& config);

        virtual ~MyReceiver();

    private: // Raw data unpacking

        size_t getDetectorSize();
        std::vector<unsigned long long> getDisplayShape();
        std::vector<unsigned long long> getDaqShape(unsigned short framesperTrain);
        void unpackRawData(const char* data, size_t idx, unsigned short* adc,␣
↪unsigned char* gain);

    };

} /* namespace karabo */

#endif /* KARABO_MYRECEIVER_HH */
```

You probably don't need anything more than that.

Some functions are pure virtual in SlsReceiver and must be defined in the derived class:

- getDetectorSize
- getDisplayShape
- getDaqShape
- unpackRawData

9.3 MyReceiver.cc file

The pure virtual functions which must be defined in the derived class are:

`size_t getDetectorSize()`
: returns the size of the detector (i.e. the number of channels, or pixels).

std::vector<unsigned long long> getDisplayShape()
returns the shape of one frame (can be 1- or 2-d).

std::vector<unsigned long long> getDaqShape(unsigned short framesPerTrain)
returns the shape of the data as needed by the DAQ (frames are grouped per train before being sent to the DAQ, therefore it is 2- or 3-d; moreover the DAQ wants the first dimension to be the fastest changing, the last dimension the slowest).

void unpackRawData(const char* data, size_t idx, unsigned short* adc, unsigned short* gain)
fill-up the <adc> and <gain> buffers with the ADC and gain values contained in <data> for the packet <idx>.

An example of MyReceiver.cc is the following. In the best case you will just have to change the constants (here for the Gotthard) to match the raw data format of the detector:

```
#include "MyReceiver.hh"

USING_KARABO_NAMESPACES

// e.g. Gotthard channels
#define MY_CHANNELS 1280

// e.g. Gotthard raw data: unpacking adc/gain bytes
#define MY_ADC_MASK       0x3FFF
#define MY_GAIN_MASK      0xC000
#define MY_GAIN_OFFSET        14

namespace karabo {

    KARABO_REGISTER_FOR_CONFIGURATION(BaseDevice, Device<>, SlsReceiver,
        MyReceiver)

    void MyReceiver::expectedParameters(Schema& expected) {
        Schema displayData;

        // This is the schema for data display in the GUI,
        // in this example for 1-d data
        NODE_ELEMENT(displayData).key("data")
                .displayedName("Data")
                .commit();

        VECTOR_UINT16_ELEMENT(displayData).key("data.adc")
                .displayedName("ADC")
                .description("The ADC counts.")
                .readOnly()
                .commit();

        VECTOR_UINT(_ELEMENT(displayData).key("data.gain")
                .displayedName("Gain")
                .description("The ADC gain.")
                .readOnly()
                .commit();

        OUTPUT_CHANNEL(expected).key("display")
                .displayedName("Display")
                .dataSchema(displayData)
                .commit();
    }

    MyReceiver::MyReceiver(const karabo::util::Hash& config) :
```

(continues on next page)

(continued from previous page)

```
            SlsReceiver(config) {
    }

    MyReceiver::~MyReceiver() {
    }

    unsigned short MyReceiver::getDetectorSize() {
        return MY_CHANNELS;
    }

    std::vector<unsigned long long> MyReceiver::getDisplayShape() {
        return {this->getDetectorSize()};
    }

    std::vector<unsigned long long> MyReceiver::getDaqShape(unsigned short␣
↪framesPerTrain) {
        // DAQ first dimension is fastest changing one
        return {this->getDetectorSize(), framesPerTrain};
    }

    void MyReceiver::unpackRawData(const char* data, size_t idx, unsigned short* adc,␣
↪unsigned char* gain) {
        // e.g. For Gotthard:
        const size_t frameSize = this->getDetectorSize();
        size_t offset = sizeof(unsigned short) * idx * frameSize;
        const char* ptr = data + offset; // Base address of the <idx> frame

        for (size_t i = 0; i < frameSize; ++i) {
            adc[i] = (reinterpret_cast<const unsigned short*> (ptr))[i] & MY_ADC_MASK;
            gain[i] = ((reinterpret_cast<const unsigned short*> (ptr))[i] & MY_GAIN_
↪MASK) >> MY_GAIN_OFFSET;
        }
    }

} /* namespace karabo */
```

9.4 Simulation Mode

To compile the slsReceiver in simulation mode, just run

```
make CONF=Simulation
```

This way the package will be linked against the simulation, instead of libSlsReceiver.

For more details on how the simulation is implemented, see *The slsDetectorSimulation* Section.

CHAPTER 10

Troubleshooting

10.1 The control device is in UNKNOWN state

This means that the Karabo device cannot connect to the detector. You should check that

- the detector is connected to the network,
- and it is powered.

A possible way to verify that the detector is online is to login to the control server and use the `ping` command[1].

Detector power can often be controlled via a Beckhoff digital output device, with the same domain name as the detector, but different type and member. For example to the detector `SA1_XTD9_HIREX/DET/GOTTHARD` correspond the Beckhoff devices `SA1_XTD9_HIREX/DCTRL/GOTTHARD_MASTER_POWER` and `SA1_XTD9_HIREX/DCTRL/GOTTHARD_SLAVE_POWER`, which can be used to power on and off the Gotthard's master and slave.

If the detector is online but still in UNKNOWN state, it can be that the server software on the detector is not running. In this you can can try to reboot the detectors micro-controller as explained in the *next* Section.

10.2 Rebooting the detector

If the detector is online but not working properly, it can be restarted by connecting to it. For example, if `192.168.194.82` is the IP of module to be restarted:

```
telnet 192.168.194.82
```

and then execute:

```
reboot
```

This command will restart the micro-controller, without the need of a complete power cycle of the detector.

[1] https://linux.die.net/man/8/ping

10.3 The control device is in ERROR state

The control device will go to ERROR state if the receiver device(s) is (are) not running. This can happen because the instantiation sequence was not correct, or because the receiver device(s) went down.

The correct starting sequence is

- start receiver device(s) first;
- then start the control device.

To recover for the ERROR, please make sure that the receiver device(s) are online, then *reset* the control device.

10.4 The control device is in INIT state

This usually means that the detector is online but cannot be configured. Try to reboot is as described in *this* Section.

10.5 The receiver device is in ERROR state

This usually means that the RX TCP port used by this Karabo device, is already in use.

10.6 The control device is ACQUIRING but receiver's *Frame Rate In* is 0

If you are in external trigger mode (*Timing Mode* = *trigger*), it is possible that the detector receives no trigger signal. You can test it by setting the trigger mode to internal (*auto*).

10.7 *Frame Rate Out* is not 0, but no images are visible in the GUI

First check that the flag *onlineDisplayEnable* on the receiver device is enabled.

If the flag is set to `True`, then it could be that the GUI server is malfunctioning. In case there is a second GUI server available for the topic, try to switch to that one.

10.8 *Frame Rate Out* is not 0, but the DAQ does not save any data

Check that in the *DAQ Output* node the *hostname* is set to the IP address of the 10 GbE interface dedicated to the DAQ.

10.9 The receiver device prints out TCP socket errors

If the receiver device logs messages like

they can be safely ignored.

This is because the control device, in order to check that the receiver is online, opens a TCP connection to it. The receiver complains as no data is exchanged before the connection is closed.

This is a periodic check repeated every 20 s.

10.10 If nothing else helps...

In order to cleanly restart the system, follow these steps:

- shutdown the Karabo devices and the servers;
- power off the detector;
- optionally execute `sls_detector_get free` on the control server (this step should not be needed, as this "free" action is done by the Karabo control device);
- power up the detector;
- instantiate the receiver device(s);
- instantiate the control device.

CHAPTER 11

The slsDetectorSimulation

In this package a subset of the slsDetector package have been reimplemented, in a simulation mode.

11.1 The Receiver Class

It reimplements part of the functionalities of the same class from the slsDetectorPackage. In order to use it, you need to include *slssimulation/Receiver.h* and link *libSlsSimulation.so*.

The *Receiver* class runs a TCP/IP server on the port specified on start-up by the argument *–rx_tcpport* or *-t*.

The commands known to the *Receiver* TCP/IP server are:

- *start*: to start generating simulated data (till the *stop* command is received);
- *stop*: to stop generating data;
- *exptime* [ns]: to set the exposure time parameter;
- *delay* [ns]: to set the delay after trigger parameter;
- *period* [ns]: to set the period parameter;
- *detectortype*: to set the detector type; an integer value must be used; the options are defined in *slssimulation/sls_simulation_defs.h*;
- *fpath*: to set the output path;
- *fname*: to set the root name of the output file;
- *findex*: to set the start index of the output file;
- *fwrite*: to enable or disable the output file write;
- *settings*: to select the settings; an integer value must be used; the options are defined in *slssimulation/sls_simulation_defs.h*.

11.2 The Detector Class

It reimplements part of the functionalities of the same class from the slsDetectorPackage. In order to use it, you need to include *slssimulation/Detector.h* and link *libSlsSimulation.so*.

It currently has the limitation that it can only control one simulated detector.

The *Detector::start()* function will connect to the *Receiver* TCP/IP server, configure it and start an acquisition.

The *Detector::stop()* function will stop the acquisition, if still ongoing.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search