libxsp

API Reference

Version 2.1

Date 2022-05-17

libxsp: API Reference

X-Spectrum GmbH

Version 2.1
Date 2022-05-17

Copyright © 2019-2022 X-Spectrum GmbH

Table of Contents

1. Free Functions

The following free functions are defined in namespace `xsp`:

`clearLogHandler()`	removes a log message handler
`createSystem()`	returns pointer to a detector system object
`libraryMajor()`	returns major version number
`libraryMinor()`	returns minor version number
`libraryPatch()`	returns patch version number
`libraryVersion()`	returns version string
`setLogHandler()`	adds a log message handler

1.1. clearLogHandler()

```
void xsp::clearLogHandler()
```

Removes the log handler, which has been set with `setLogHandler()`.

Important

This function has to called before the handler goes out of scope.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    xsp::setLogHandler([&](xsp::LogLevel l, const std::string& m) {
            // handle log
        });

    // run acquisition

    xsp::clearLogHandler();
    return 0;
}
```

1.2. createSystem()

```
std::unique_ptr<xsp::System> xsp::createSystem(const std::string& config_file)
```

Creates a `System` object, initialized with values from the specified configuration file.

The function may return a nullptr, if multi-host synchronization could not be established. This is usually the case, if the process receives a SIGINT signal (e.g. by pressing Ctrl-C) while still in the synchronizing phase.

The function throws a `ConfigError` exception, if the configuration file cannot be opened for reading, or if there were errors while reading.

Parameters

config_file	absolute path of configuration file

Return Value

A pointer to a `System` object

Example

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");

    return 0;
}
```

1.3. libraryMajor()

```
int xsp::libraryMajor()
```

Returns the major version number of the library.

Return Value

The major version number

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    std::cout << “libxsp major: “ << xsp::libraryMajor() << std::endl;

    return 0;
}
```

Possible Output

```
libxsp major: 2
```

1.4. libraryMinor()

```
int xsp::libraryMinor()
```

Returns the minor version number of the library.

Return Value

The minor version number

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    std::cout << “libxsp minor: “ << xsp::libraryMinor() << std::endl;

    return 0;
}
```

Possible Output

```
libxsp minor: 0
```

1.5. libraryPatch()

```
int xsp::libraryPatch()
```

Returns the patch version number of the library.

Return Value

The patch version number

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    std::cout << “libxsp patch: “ << xsp::libraryPatch() << std::endl;

    return 0;
}
```

Possible Output

```
libxsp patch: 2
```

1.6. libraryVersion()

```
std::string xsp::libraryVersion()
```

Returns the version of the library as a string in the format "major.minor.patch".

Return Value

A string with the library version

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    std::cout << “libxsp version: “ << xsp::libraryVersion() << std::endl;

    return 0;
}
```

Possible Output

```
libxsp version: 1.0.2
```

1.7. setLogHandler()

```
void xsp::setLogHandler(
        const std::function<void(LogLevel, const std::string&)>& h
        )
```

Sets a handler, which is called to log information from within the library. The handler must be defined as a method to accept two arguments: the level of type `LogLevel` and a string.

Important

It must be assured that the handler is callable until `clearLogHandler()` is called.

Parameters

h	a callable object (e.g. a lambda expression)

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    xsp::setLogHandler([level](xsp::LogLevel l, const std::string& m) {
            switch (l) {
                case xsp::LogLevel::ERROR:
                    std::cerr << l << ": " << m << std::endl;
                    break;
                default:
                    std::cout << l << ": " << m << std::endl;
                    break;
            }
        });

    auto s = xsp::createSystem("/path/to/system.yml");

    xsp::clearLogHandler();
    return 0;
}
```

Possible Output

```
DEBUG: created DetectorSystem with ID SYS
```

2. Common Classes

The following common classes are defined in namespace `xsp`:

`ConfigError`	represents a configuration error
`Detector`	base class for detectors
`Error`	base class for all exceptions
`EventType`	type of user event
`Frame`	acquired frame
`FrameStatusCode`	status code of acquired frame
`LogLevel`	log severity level
`Position`	module position
`PostDecoder`	base class for post decoder
`Receiver`	base class for receivers
`Rotation`	module rotation
`RuntimeError`	indicates a runtime error
`ShuffleMode`	shuffle mode used for compression
`StatusCode`	status code of a detector operation
`System`	detector system

2.1. ConfigError

Represents an error while parsing the system configuration file.

It inherits the public member methods from `Error`.

2.2. Detector

Represents the control interface to a physical detector.

It provides the following public member methods:

`clearEventHandler()`	removes a user event handler
`connect()`	connects a detector
`disconnect()`	disconnects a detector
`frameCount()`	returns number of frames to acquire
`id()`	returns the detector ID
`initialize()`	initializes a detector
`isBusy()`	returns whether a detector is busy with acquisition
`isConnected()`	returns whether a detector is connected
`isReady()`	returns whether a detector is ready for acquisition
`liveFrame()`	returns pointer to actual live frame
`liveFrameDepth()`	returns bit depth of live frame
`liveFrameHeight()`	returns live frame height
`liveFrameSelect()`	selects live frame to display
`liveFramesQueued()`	returns number of queued live frames
`liveFrameWidth()`	returns live frame width
`release()`	releases a live frame from the queue
`reset()`	resets a detector
`setEventHandler()`	adds a user event handler
`setFrameCount()`	sets number of frames to acquire
`setShutterTime()`	sets shutter time

`shutterTime()`	returns shutter time
`startAcquisition()`	starts an acquisition
`stopAcquisition()`	stops an acquisition
`type()`	returns the detector type
`userData()`	returns a user data value

2.2.1. clearEventHandler()

```
void xsp::Detector::clearEventHandler()
```

Removes handler, which has been previously set with `setEventHandler()`.

Important

This function has to called before the handler goes out of scope.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setEventHandler([&](auto t, const void* d) {
            // handle event
        });

    // perform acquisition

    d->clearEventHandler();
    return 0;
}
```

2.2.2. connect()

```
void xsp::Detector::connect()
```

Opens the control connection to the detector. Depending on the detector type, some initial data transfer is performed to identify the connected detector modules.

If connection cannot be established, a `RuntimeError` exception with status code `BAD_COMMUNICATION_ERROR` is thrown. For a multi-module detector, a failure on at least one module will disconnect the whole detector unless the `ModuleFlag` is set on that module.

If the connection can be established, but identification of a module failed, a `RuntimeError` exception with status code `BAD_DEVICE_FAILURE` is thrown. The detector is not disconnected, so that communication to other modules, which have been identified correctly, is still possible.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->connect();
```

```
    return 0;
}
```

2.2.3. disconnect()

```
void xsp::Detector::disconnect()
```

Closes the control connection to the detector.

Calling this method has no effect, if the detector is already disconnected.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->connect();
    d->disconnect();

    return 0;
}
```

2.2.4. frameCount()

```
std::uint64_t xsp::Detector::frameCount() const
```

Returns the currently configured number of frames to acquire.

The frame count is not read from detector hardware, but instead is cached within the library. It is therefore necessary to call `setFrameCount()` at least once to get a meaningful value.

Return Value

Number of frames to acquire

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setFrameCount(100);
    std::cout << "n frames: " << d->frameCount() << std::endl;

    return 0;
}
```

Output

```
n frames: 100
```

2.2.5. id()

```
std::string xsp::Detector::id() const
```

Returns the detector ID.

The detector ID is configured as:

```
detectors:
  - id: DET_01
    […]
```

Return Value

A string with the detector ID

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    std::cout << "detector ID: " << d->id() << std::endl;

    return 0;
}
```

Possible Output

```
detector ID: DET_01
```

2.2.6. initialize()

```
void xsp::Detector::initialize()
```

Initializes the detector.

This function requires that the detector has been connected by calling `connect()` beforehand. If the detector is disconnected, then this function throws a `RuntimeError` exception with status code BAD_NOT_CONNECTED.

The exact behavior of this function depends on the detector model, but usually initial configuration and calibration data are sent to the detector. If the communication with the detector fails, a `RuntimeError` exception with status code BAD_COMMUNICATION_ERROR is thrown.

Execution of this function takes some time to transfer data into the detector.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->connect();
    d->initialize();

    return 0;
}
```

2.2.7. isBusy()

```
bool xsp::Detector::isBusy()
```

Returns whether a detector is busy.

A detector is busy, if `startAcquisition()` has been called. This flag is reset, when `stopAcquisition()` is called or when all frames have been received.

Users might check detector state using `isBusy()` before calling `startAcquisition()` to be sure that no acquisition is currently ongoing.

Return Value

True, if detector is busy with acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto d = s->detector("DET_01");

    if (!d->isBusy())
        d->startAcquisition();
    std::cout << "detector DET_01 busy: "
        << (d->isBusy() ? "yes" : "no") << std::endl;

    return 0;
}
```

Output

```
detector DET_01 busy: yes
```

2.2.8. isConnected()

```
bool xsp::Detector::isConnected()
```

Returns whether a detector is connected.

A detector is connected, if the control network connection has been established.

Return Value

True, if a detector is connected

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    s->connect();
    if (d->isConnected())
        std::cout << "detector DET_01 connected" << std::endl;

    return 0;
}
```

Possible Output

```
detector DET_01 connected
```

2.2.9. isReady()

```
bool xsp::Detector::isReady()
```

Returns whether a detector is ready.

A detector is ready, if the detector and his associated data receivers have been successfully initialized. The detector is then ready for data acquisition and `startAcquisition()` can be called.

Return Value

True, if a detector is ready for acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    s->connect();
    s->initialize();

    while (!d->isReady())
        sleep(1);
    std::cout << "detector DET_01 ready" << std::endl;

    return 0;
}
```

Possible Output

```
detector DET_01 ready
```

2.2.10. liveFrame()

```
const xsp::Frame* xsp::Detector::liveFrame(int timeout_ms) const
```

Returns a pointer to the live frame.

The live frame is the composition of all module frames, and is updated regularly during acquisition. Frame data is guaranteed to be valid until `release()` has been called.

The update period can be changed in the `detectors:` section of the system configuration file, as shown in the following example:

```
detectors:
  - id: DET_01
    type: Lambda
    live-update: 1000   # ms
    [...]
```

The default update period is 1000 ms.

Note

The update period is translated into a frame number difference. Thus the actual time between two live updates depends on the time required to decode all intermediate frames. This time can be considerably longer than the configured update period, especially at short shutter times.

Parameters

timeout_ms	timeout in milliseconds to wait for frames

Return Value

A pointer to a `Frame` object.

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = d->liveFrame();
    // display f->data()
    d->release(f);

    return 0;
}
```

2.2.11. liveFrameDepth()

```
int xsp::Detector::liveFrameDepth() const
```

Returns the bit depth of the live frame. The value represents a number of significant bits per frame pixel. For example, a frame depth of 12 bits indicates that frame pixels are stored as 16-bit integers, where only the lower 12 bits may have non-zero values.

The live frame is the composition of all module frames, and is updated regularly during acquisition.

Return Value

An integer with the number of valid bits per frame pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto d = s->detector("DET_01");
    std::cout << "frame depth: " << d->liveFrameDepth() << std::endl;

    return 0;
}
```

Possible Output

```
frame depth: 12
```

2.2.12. liveFrameHeight()

```
int xsp::Detector::liveFrameHeight() const
```

Returns the height of the live frame.

The live frame is the composition of all module frames, and is updated regularly during acquisition.

Return Value

An integer with the height of the frame in pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    std::cout << "frame height: " << d->liveFrameHeight() << std::endl;

    return 0;
}
```

Possible Output

```
frame height: 516
```

2.2.13. liveFrameSelect()

```
void xsp::Detector::liveFrameSelect(int subnr, int conn) const
```

Selects the frame to use as live frame based on the specified subframe and connector number. The connector number is used for discrete sensor layouts to select the live view from a specific connector, for non-discrete (i.e. compound) sensor layouts, it has to be set to 1.

In dual counter mode, subframe number 1 selects the frame with the lower threshold to be displayed in the live view, and subframe number 2 selects the frame with the higher threshold to be displayed in the live view.

For discrete sensor layout, the connector number selects the compact sensor to be displayed in the live view. The connector number must be between 1 and 4.

If liveFrameSelect() has not been called, then the default is to select subframe number 1 and connector 1.

Parameters

subnr	subframe number (either 1 or 2)
conn	connector number (between 1 and 4)

Example

```
#include <iostream>
```

```
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->liveFrameSelect(1, 1);
    d->startAcquisition();
    // display live frames

    return 0;
}
```

2.2.14. liveFramesQueued()

```
int xsp::Detector::liveFramesQueued() const
```

Returns the number of live frames that have been queued inside the detector. If this number is non-zero, users are able to get live frames by calling the `liveFrame()` method.

Return Value

Number of frames actually queued

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setFrameCount(10);
    d->setShutterTime(0.5);
    d->startAcquisition();
    usleep(10000);
    std::cout << "live frames queued: " << d->liveFramesQueued() << std::endl;

    return 0;
}
```

Possible Output

```
live frames queued: 10
```

2.2.15. liveFrameWidth()

```
int xsp::Detector::liveFrameWidth() const
```

Returns the width of the live frame.

The live frame is the composition of all module frames, and is updated regularly during acquisition.

Return Value

An integer with the width of the frame in pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    std::cout << "frame width: " << d->liveFrameWidth() << std::endl;

    return 0;
}
```

Possible Output

```
frame width: 772
```

2.2.16. release()

```
void xsp::Detector::release(const xsp::Frame* f)
```

Releases a live frame.

This function must be called after the frame data has been processed in order to remove the frame from the internal queue.

Parameters

f	pointer as returned by a previous call to `liveFrame()`

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setFrameCount(100);
    d->setShutterTime(0.5);
    d->startAcquisition();
    while (true) {
        auto f = d->liveFrame(100);
        if (f == nullptr) break;
        // display frame data
        d->release(f);
    }

    return 0;
}
```

2.2.17. reset()

```
void xsp::Detector::reset()
```

Resets a detector.

This function throws a `RuntimeError` exception, if there are communication failures while writing the reset command into the detector.

The detector is automatically reinitialized after the reset. The state of the detector after a reset is identical to the state after initialization.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    d->reset();

    return 0;
}
```

2.2.18. setEventHandler()

```
void xsp::Detector::setEventHandler(
        const std::function<void(xsp::EventType,const void*)>& h
        )
```

Sets a handler, which is called on user events.

The handler is called with 2 arguments: the `EventType` and a pointer. The pointer may point to auxiliary data, depending on the event type. If no auxiliary data is required, then the pointer must be set to `nullptr`.

Parameters

h	a callable object (e.g. a lambda expression)

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    bool ready_flag = false;
    d->setEventHandler([&](auto t, const void* d) {
            switch (t) {
                case xsp::EventType::READY:
                    std::cerr << "received READY event" << std::endl;
                    ready_flag = true;
                    break;
            }
        });

    s->connect();
    s->initialize();
    while (!ready_flag)
        sleep(1);
    std::cout << "detector ready" << std::endl;
```

```
    d->clearEventHandler();
    return 0;
}
```

Possible Output

```
received READY event
detector ready
```

2.2.19. setFrameCount()

```
void xsp::Detector::setFrameCount(uint64_t count)
```

Sets the number of frames to acquire to count.

This function throws a `RuntimeError` exception, if there are communication failures while writing the value into the detector.

Parameters

count	number of frames to acquire

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setFrameCount(100);
    std::cout << "n frames: " << d->frameCount() << std::endl;

    return 0;
}
```

Output

```
n frames: 100
```

2.2.20. setShutterTime()

```
void xsp::Detector::setShutterTime(double time_ms)
```

Sets the shutter time.

This function throws a `RuntimeError` exception, if there are communication failures while writing the value into the detector.

The valid value range depends on the specific detector type.

Parameters

time_ms	shutter time in milliseconds

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
```

```
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setShutterTime(0.5);
    std::cout << "shutter time [ms]: " << d->shutterTime() << std::endl;

    return 0;
}
```

Output

```
shutter time [ms]: 0.5
```

2.2.21. shutterTime()

```
double xsp::Detector::shutterTime() const
```

Returns the currently configured shutter time in milliseconds.

The shutter time is not read from detector hardware, but instead is cached within the library. It is therefore necessary to call `setShutterTime()` at least once to get a meaningful value.

Return Value

The shutter time in milliseconds

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setShutterTime(0.5);
    std::cout << "shutter time [ms]: " << d->shutterTime() << std::endl;

    return 0;
}
```

Output

```
shutter time [ms]: 0.5
```

2.2.22. startAcquisition()

```
void xsp::Detector::startAcquisition()
```

Starts an acquisition.

The method can only be called, if the detector is not busy. If it is called, while the detector is busy, a `RuntimeError` exception with status code `BAD_DEVICE_BUSY` is thrown.

This function throws a `RuntimeError` exception with status code `BAD_COMMUNICATION_ERROR`, if there are communication failures while writing the start command into the detector.

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    d->setFrameCount(100);
    d->setShutterTime(0.5);
    d->startAcquisition();

    return 0;
}
```

2.2.23. stopAcquisition()

```
void xsp::Detector::stopAcquisition()
```

Stops a running acquisition.

If no acquisition is running, this function has no effect and returns immediately.

This function throws a `RuntimeError` exception, if there are communication failures while writing the start command into the detector.

Execution of this functions may take at most the time, that has been configured as link timeout (in milliseconds) for receivers.[1]

```
receivers:
  - […]
    links:
      - […]
        timeout: 1000
```

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setFrameCount(100);
    d->setShutterTime(1000.0);
    d->startAcquisition();
    sleep(5);
    d->stopAcquisition();

    return 0;
}
```

2.2.24. type()

```
std::string xsp::Detector::type() const
```

Returns the detector type.

The detector type is configured as:

```
detectors:
  - id: DET_01
```

[1]Internally, the link timeout is used to regularly check, whether an acquisition stop has been requested.

```
    type: Lambda
```

Return Value

A string with the detector type

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    std::cout << "detector type: " << d->type() << std::endl;

    return 0;
}
```

Possible Output

```
detector type: Lambda
```

2.2.25. userData()

```
std::string xsp::Detector::userData(const std::string& key)
```

Returns configured user data value associated with requested key.

The method returns an empty string, if a key cannot be found.

Parameters

key	the key for a user data item

Return Value

A string representing the value of a user data key

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    std::cout << "sensor material: "
        << d->userData("sensor_material") << std::endl;

    return 0;
}
```

Possible Output

```
sensor material: Si
```

2.3. Error

Base class for all library exceptions: `ConfigError`, `RuntimeError`.

It provides the following public member methods:

`what()`	returns a textual description of the error condition

2.3.1. what()

```
const char* xsp::Error::what() const
```

Returns the textual description of the error.

2.4. EventType

This enumeration indicates the type of an event.

Values

READY	detector or receiver is ready
START	detector has been started
STOP	detector has been stopped

2.5. Frame

This class represents an acquired frame.

It provides the following public member methods:

`connector()`	returns the connector number
`data()`	returns pointer to the frame data
`nr()`	returns the frame number
`seq()`	returns the hardware generated sequence number
`size()`	returns the number of frame data bytes
`status()`	returns the frame status code
`subframe()`	returns the subframe number
`trigger()`	returns the trigger number

2.5.1. connector()

```
int xsp::Frame::connector() const
```

Returns the connector number for Hydra type detectors with discrete sensors. For detectors with compound sensors, the connector number is always 1.

Return Value

The connector number

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
```

```
    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
         std::cout << "received frame #" << f->nr()
             << " on connector #" << f->connector() << std::endl;
         r->release(f);
    }

    return 0;
}
```

Possible Output

```
received frame #1 on connector #1
```

2.5.2. data()

```
const std::uint8_t* xsp::Frame::data() const
```

Returns a pointer to the frame data. The size of the array can be determined with the `size()` method.

Frame data is returned as a pointer to a byte array. The exact meaning of each byte depends on the detector type and operation mode. It might be a chunk of compressed data, a two-dimensional array of pixels, or a record with a detector specific structure.

The pointer may need to be casted to a different integer size in order to access multi-byte values.

Return Value

A pointer to a byte array

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
        // assuming frame data is a 2D array of 16-bit uints
        auto dp = reinterpret_cast<std::uint16_t*>(f->data());
        std::cout << “pixel value at origin: “ << dp[0];
        r->release(f);
    }

    return 0;
}
```

Possible Output

```
pixel value at origin: 42
```

2.5.3. nr()

```
std::uint64_t xsp::Frame::nr() const
```

Returns the frame number.

The number will start at 1 for each new acquisition.

Return Value

Frame number

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
         std::cout << "received frame #" << f->nr() << std::endl;
         r->release(f);
    }

    return 0;
}
```

Possible Output

```
received frame #1
```

2.5.4. seq()

```
std::uint64_t xsp::Frame::seq() const
```

Returns the frame sequence number, as generated by the hardware.

The number may have less than 64 significant bits. It wraps to zero on overflow of the significant bits. It is not guaranteed that the detector resets the number when starting a new acquisition run.

Return Value

Frame sequence number

Example

```
#include <iostream>
#include <libxsp.h>
```

```
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
         std::cout << "received frame #" << f->nr()
             << ", seq #" << f->seq() << std::endl;
         r->release(f);
    }

    return 0;
}
```

Possible Output

```
received frame #1, seq #177
```

2.5.5. size()

```
std::size_t xsp::Frame::size() const
```

Returns the size of the frame data array in number of bytes.

Return Value

Size of the frame data array in bytes

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
        std::cout << “frame size [bytes]: “ << f->size();
        r->release(f);
    }

    return 0;
}
```

Possible Output

```
frame size [bytes]: 796704
```

2.5.6. status()

```
xsp::FrameStatusCode xsp::Frame::status() const
```

Returns the frame status code of type `FrameStatusCode`.

Return Value

Frame status code

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
        if (f->status() == xsp::FrameStatusCode::FRAME_OK)
            std::cout << "frame is ok" << std::endl;
        else
            std::cout << "frame is not ok" << std::endl;
        r->release(f);
    }

    return 0;
}
```

Possible Output

```
frame is ok
```

2.5.7. subframe()

```
int xsp::Frame::subframe() const
```

Returns the subframe number.

In single counter mode, the subframe number is always 1. In dual counter mode, subframe 1 contains the counter values for the lower threshold and subframe 2 contains the counter values for the higher threshold.

Return Value

The subframe number

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
```

```
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
         std::cout << "received frame #" << f->nr()
             << ", subframe #" << f->subframe() << std::endl;
         r->release(f);
    }

    return 0;
}
```

Possible Output

```
received frame #1, subframe #1
```

2.5.8. trigger()

```
std::uint64_t xsp::Frame::trigger() const
```

Returns the trigger pulse number.

The trigger pulse number is generated inside the detector by incrementing a counter for each new trigger. Older detectors may not count trigger pulses, in that case, a value of 0 is returned.

This value may overflow and restart at 0, if the counter inside the detector has less than 64 bit. It is not guaranteed that the detector resets the counter when starting a new acquisition run.

Return Value

The trigger pulse number

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    if (f != nullptr) {
        std::cout << "received frame #" << f->nr()
            << " on trigger pulse " << f->trigger() << std::endl;
        r->release(f);
    }

    return 0;
```

```
}
```

Possible Output

```
received frame #17 on trigger pulse 2
```

2.6. FrameStatusCode

This enumeration indicates the status of a received frame.

Values

FRAME_OK	the frame contains valid data
FRAME_INCOMPLETE	the frame has been received incompletely due to loss of packets during transmission
FRAME_MISSING	the frame has not been received at all due to loss of packets during transmission
FRAME_COMPRESSION_FAILED	the frame could not be compressed

2.7. LogLevel

This enumeration indicates the severity level of a log message. It can have the following values:

ERROR	for critical messages, current task is aborted
WARN	for non-critical messages, current task can continue
INFO	for informal messages
DEBUG	for more verbose informal messages

2.8. Position

This struct represents a position in terms of three coordinates.

Members

x	x coordinate
y	y coordinate
z	z coordinate

2.9. PostDecoder

The PostDecoder class represents the interface to an additional post processing block, which further processes the decoded images. This includes summing of several consecutive images, and stitching images from multiple modules into a single image for the whole detector.

It provides the following public member methods:

`clearEventHandler()`	removes a user event handler
`compressionEnabled()`	returns whether compression is enabled
`compressionLevel()`	returns current compression level
`compressor()`	returns configured compressor
`frame()`	returns pointer to next processed frame
`frameDepth()`	returns bit depth of processed frame
`frameHeight()`	returns height of processed frame
`framesQueued()`	returns number of queued frames
`frameSummingEnabled()`	returns whether frame summing is enabled

`frameWidth()`	returns width of processed frame
`id()`	returns the id
`initialize()`	initializes the post decoder
`isBusy()`	returns whether a post decoder is busy with acquisition
`isReady()`	returns whether a post decoder is ready for acquisition
`numberOfSubFrames()`	returns the number of subframes per exposure
`numberOfConnectedSensors()`	returns the number of connected sensors
`release()`	releases an processed frame from the queue
`setCompressionLevel()`	sets the compression level
`setEventHandler()`	adds an user event handler
`setShuffleMode()`	sets shuffle mode for compression
`setSummedFrames()`	sets number of frames which are summed
`shuffleMode()`	returns current shuffle mode for compression
`summedFrames()`	returns number of frames which are summed

2.9.1. clearEventHandler()

```
void xsp::PostDecoder::clearEventHandler()
```

Removes an event handler, which has been previously set with `setEventHandler()`.

Important

This function has to called before the handler goes out of scope.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    p->setEventHandler([&](auto t, const void* d) {
            // handle event
        });

    // perform acquisition

    p->clearEventHandler();
    return 0;
}
```

2.9.2. compressionEnabled()

```
bool xsp::PostDecoder::compressionEnabled() const
```

Returns whether compression is enabled. The compression is enabled via system configuration file by setting the compressor in the post decoding block to something else than "none".

Return Value

True, if compression is enabled

Example

```
#include <iostream>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto p = s->postDecoder("DET_01");

    std::cout << "compression: "
        << p->compressionEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Possible Output

```
compression: enabled
```

2.9.3. compressionLevel()

```
int xsp::PostDecoder::compressionLevel() const
```

Returns the configured compression level.

Return Value

Compression level

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto p = s->postDecoder("DET_01");
    std::cout << "compression level: " << p->compressionLevel() << std::endl;

    return 0;
}
```

Possible Output

```
compression level: 4
```

2.9.4. compressor()

```
std::string xsp::PostDecoder::compressor() const
```

Returns the configured compressor. If no compressor has been configured, then the string "none" is returned.

Return Value

A string with the configured compressor

Example

```
#include <iostream>
#include <libxsp.h>

int main()
```

```
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto p = s->postDecoder("DET_01");

    std::cout << "compressor: " << p->compressor() << std::endl;

    return 0;
}
```

Possible Output

```
compressor: zlib
```

2.9.5. frame()

```
const xsp::Frame* xsp::PostDecoder::frame(int timeout_ms) const
```

Returns a pointer to a processed frame. The function waits the specified time in milliseconds for a frame to be queued. If no frame has been processed and queued within this time, a nullptr is returned.

Frame data is guaranteed to be valid until `release()` has been called.

> **Note**
>
> This function does not remove the frame from the queue. Thus, `release()` must be called after frame data has been processed.

In single counter mode, one frame is queued for each exposure. In dual counter mode, two frames with same frame number and sub frame numbers 1 and 2 are queued for each exposure. The number of sub frames can be determined by calling `numberOfSubFrames()`.

Parameters

timeout_ms	timeout in milliseconds to wait for frames

Return Value

A pointer to a `Frame` object or a nullptr.

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto p = s->postDecoder("DET_01");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = p->frame(100);
    p->release(f);

    return 0;
```

```
}
```

2.9.6. frameDepth()

```
int xsp::PostDecoder::frameDepth() const
```

Returns the bit depth of the frame. The value represents a number of significant bits per frame pixel. For example, a frame depth of 12 bits indicates that frame pixels are stored as 16-bit integers, where only the lower 12 bits may have non-zero values.

The frame depth is only available after the system has been initialized.

Return Value

Number of valid bits per frame pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    std::cout << "frame depth: " << p->frameDepth() << std::endl;

    return 0;
}
```

Possible Output

```
frame depth: 12
```

2.9.7. frameHeight()

```
int xsp::PostDecoder::frameHeight() const
```

Returns the height of the frame.

The frame height is only available after the system has been initialized.

Return Value

Height of the frame in pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    std::cout << "frame height: " << p->frameHeight() << std::endl;

    return 0;
```

```
}
```

Possible Output

```
frame height: 1163
```

2.9.8. framesQueued()

```
int xsp::PostDecoder::framesQueued() const
```

Returns the number of processed frames inside the internal frame queue. If this number is non-zero, users are able to get frames by calling the `frame()` function.

Return Value

Number of frames actually queued

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setFrameCount(10);
    d->setShutterTime(0.5);
    d->startAcquisition();
    usleep(10000);
    std::cout << "frames queued: " << p->framesQueued() << std::endl;

    return 0;
}
```

Possible Output

```
frames queued: 10
```

2.9.9. frameSummingEnabled()

```
bool xsp::PostDecoder::frameSummingEnabled() const
```

Returns if frame summing is enabled. Frame summing is enabled, if the number of summed frames is greater than 1.

Return Value

True if frame summing is enabled, false otherwise.

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
```

```
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    std::cout << "frame summing enabled: " << p->frameSummingEnabled() \
        << std::endl;

    return 0;
}
```

Possible Output

```
frame summing enabled: true
```

2.9.10. frameWidth()

```
int xsp::PostDecoder::frameWidth() const
```

Returns the width of the frame.

The frame width is only available after the system has been initialized.

Return Value

Width of the frame in pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");

    s->connect();
    s->initialize();
    std::cout << "frame width: " << p->frameWidth() << std::endl;

    return 0;
}
```

Possible Output

```
frame width: 800
```

2.9.11. id()

```
std::string xsp::PostDecoder::id() const
```

Returns the post decoder ID.

The post decoder ID is equivalent to the configured reference, detector ID or detector ID plus '/' plus module number, configured as:

```
postdecoders:
  - ref: DET_01/1
    […]
  # or
  - ref: DET_01
    […]
```

Return Value

A string with the post decoder ID

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    std::cout << "post decoder ID: " << p->id() << std::endl;

    return 0;
}
```

Possible Output

```
post decoder ID: DET_01
```

2.9.12. initialize()

```
void xsp::PostDecoder::initialize()
```

Initializes a post decoder.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    p->initialize();

    return 0;
}
```

2.9.13. isBusy()

```
bool xsp::PostDecoder::isBusy() const
```

Returns whether a post decoder is busy.

A post decoder is busy, if an acquisition has been started.

Return Value

True, if a post decoder is busy with acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    while (!p->isBusy())
        sleep(1);
    std::cout << "post decoder DET_01 busy" << std::endl;
    while (p->isBusy())
```

```
        sleep(1);
    std::cout << "post decoder DET_01 idle" << std::endl;

    return 0;
}
```

Output

```
post decoder DET_01 busy
post decoder DET_01 idle
```

2.9.14. isReady()

```
bool xsp::PostDecoder::isReady() const
```

Returns whether a post decoder is ready.

A post decoder is ready, if it has been successfully initialized. The post decoder is then ready for data reception, i.e. `frame()` can be called to access the processed frames.

Return Value

True, if a post decoder is ready for acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    while (!p->isReady())
        sleep(1);
    std::cout << "post decoder DET_01 ready" << std::endl;

    return 0;
}
```

Possible Output

```
post decoder DET_01 ready
```

2.9.15. numberOfConnectedSensors()

```
int xsp::PostDecoder::numberOfConnectedSensors() const
```

Returns the number of connected sensors in case of Hydra type Lambda detectors with discrete compact sensors. For normal detectors, such as 60K, 250K, 350K, 750K, a value of 1 is returned.

Important

Discrete sensor layout is currently not supported in the post decoder.

Return Value

Number of connected sensors

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    std::cout << "connected sensors: "
        << p->numberOfConnectedSensors() << std::endl;

    return 0;
}
```

Possible Output

```
connected sensors: 1
```

2.9.16. numberOfSubFrames()

```
int xsp::PostDecoder::numberOfSubFrames() const
```

Returns the number of subframes per exposure. This number is usually 1. For dual counter mode, 2 subframes are acquired per exposure.

For discrete sensor layout, the number of subframes is equal to the number of connected single chip sensors. If in addition dual counter mode is enabled, then twice the number of connected chip sensors is returned.

When reading the decoded frames from the post decoder, all subframes will have the same frame number, but different subframe numbers. Subframe number always starts at 1.

Return Value

Number of subframes per exposure

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    std::cout << "n subframes: "
        << p->numberOfSubFrames() << std::endl;

    return 0;
}
```

Possible Output

```
n subframes: 1
```

2.9.17. release()

```
void xsp::PostDecoder::release(const xsp::Frame* f)
```

Releases a frame from the queue.

This function must be called after the frame data has been processed in order to remove the frame from the queue.

Parameters

f	pointer to the frame as returned by a previous call to `frame()`

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
   auto p = s->postDecoder("DET_01");

    d->setFrameCount(100);
    d->setShutterTime(0.5);
    d->startAcquisition();
    while (true) {
        auto f = p->frame(100);
        if (f == nullptr) break;
        // process frame data
        p->release(f);
    }
    return 0;
}
```

2.9.18. setCompressionLevel()

```
void xsp::PostDecoder::setCompressionLevel(int level) const
```

Returns the configured compression level.

Return Value

Compression level

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto p = s->postDecoder("DET_01");
    p->setCompressionLevel(1);
    std::cout << "compression level: " << p->compressionLevel() << std::endl;

    return 0;
}
```

Possible Output

```
compression level: 1
```

2.9.19. setEventHandler()

```
void xsp::PostDecoder::setEventHandler(
        const std::function<void(xsp::EventType,const void*)>& h
        )
```

Sets a handler, which is called on user events.

The handler is called with 2 arguments: the `EventType` and a pointer. The pointer may point to auxiliary data, depending on the event type. If no auxiliary data is required, then the pointer must be set to `nullptr`.

Parameters

h	a callable object (e.g. a lambda expression)

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    bool started_flag = false;
    p->setEventHandler([&](auto t, const void* d) {
            switch (t) {
                case xsp::EventType::START:
                    std::cout << "received START event" << std::endl;
                    ready_flag = true;
                    break;
            }
        });

    s->connect();
    s->initialize();
    d->startAcquisition();

    while (!started_flag)
        sleep(1);
    std::cout << "postDecoder started" << std::endl;

    p->clearEventHandler();
    return 0;
}
```

Possible Output

```
received START event
postDecoder started
```

2.9.20. setShuffleMode()

```
void xsp::PostDecoder::setShuffleMode(xsp::ShuffleMode) const
```

Returns the configured `ShuffleMode`.

Return Value

Enumeration of type `ShuffleMode`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
```

```
    s->connect();
    auto p = s->postDecoder("DET_01");

    p->setShuffleMode(xsp::ShuffleMode::NO_SHUFFLE);
    switch (p->shuffleMode()) {
     case xsp::ShuffleMode::NO_SHUFFLE:
       std::cout << "shuffle mode: NO_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BYTE_SHUFFLE:
       std::cout << "shuffle mode: BYTE_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BIT_SHUFFLE:
       std::cout << "shuffle mode: BIT_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::AUTO_SHUFFLE:
       std::cout << "shuffle mode: AUTO_SHUFFLE" << std::endl;
       break;
    }

    return 0;
}
```

Possible Output

```
shuffle mode: NO_SHUFFLE
```

2.9.21. setSummedFrames()

```
int xsp::PostDecoder::setSummedFrames() const
```

Sets the number of frames, which are summed into a single combined frame. A value of 1 disables the frame summing.

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    p->setSummedFrames(100);
    std::cout << "summed frames: " << p->summedFrames() << std::endl;
    if (p->frameSummingEnabled()) {
        std::cout << "frame summing enabled" << std::endl;
    } else {
        std::cout << "frame summing disabled" << std::endl;
    }

    return 0;
}
```

Possible Output

```
summed frames : 100
frame summing enabled
```

2.9.22. shuffleMode()

```
xsp::ShuffleMode xsp::PostDecoder::shuffleMode() const
```

Returns the configured `ShuffleMode`.

Return Value

Enumeration of type `ShuffleMode`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto p = s->postDecoder("DET_01");
    switch (p->shuffleMode()) {
     case xsp::ShuffleMode::NO_SHUFFLE:
       std::cout << "shuffle mode: NO_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BYTE_SHUFFLE:
       std::cout << "shuffle mode: BYTE_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BIT_SHUFFLE:
       std::cout << "shuffle mode: BIT_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::AUTO_SHUFFLE:
       std::cout << "shuffle mode: AUTO_SHUFFLE" << std::endl;
       break;
    }

    return 0;
}
```

Possible Output

```
shuffle mode: AUTO_SHUFFLE
```

2.9.23. summedFrames()

```
int xsp::PostDecoder::summedFrames() const
```

Returns number of frames, which are summed into a single frame. A value of 1 disables the frame summing.

Return Value

Number of summed frames

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto p = s->postDecoder("DET_01");
    s->connect();
    s->initialize();

    std::cout << "summed frames: " << p->summedFrames() << std::endl;

    return 0;
}
```

Possible Output

```
summed frames : 10
```

2.10. Receiver

The Receiver class represents the data interface to a physical detector.

It provides the following public member methods:

clearEventHandler()	removes a user event handler
connect()	connects a receiver
disconnect()	disconnects a receiver
frame()	returns pointer to next acquired frame
frameDepth()	returns bit depth of acquired frame
frameHeight()	returns height of acquired frame
framesQueued()	returns number of queued frames
frameWidth()	returns width of acquired frame
id()	returns the receiver ID
initialize()	initializes a receiver
isBusy()	returns whether a receiver is busy with acquisition
isConnected()	returns whether a receiver is connected
isReady()	returns whether a receiver is ready for acquisition
ramAllocated()	returns whether RAM buffer has been completely allocated
release()	releases an acquired frame from the queue
setEventHandler()	adds a user event handler
type()	returns the receiver type
userData()	returns a user data value

2.10.1. clearEventHandler()

```
void xsp::Receiver::clearEventHandler()
```

Removes an event handler, which has been previously set with `setEventHandler()`.

Important

This function has to called before the handler goes out of scope.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->detector("DET_01/1");
    r->setEventHandler([&](auto t, const void* d) {
            // handle event
        });

    // perform acquisition

    r->clearEventHandler();
    return 0;
}
```

2.10.2. connect()

```
void xsp::Receiver::connect()
```

Opens the data connection to the receiver.

If connection cannot be established, a `RuntimeError` exception with status code `BAD_COMMUNICATION_ERROR` is thrown.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    r->connect();

    return 0;
}
```

2.10.3. disconnect()

```
void xsp::Receiver::disconnect()
```

Closes the data connection to the receiver.

Calling this function has no effect, if the receiver is already disconnected.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    r->connect();
    r->disconnect();

    return 0;
}
```

2.10.4. frame()

```
const xsp::Frame* xsp::Receiver::frame(int timeout_ms) const
```

Returns a pointer to an acquired frame. The function waits for the specified time in milliseconds for a frame to be queued. If no frame has been received and queued within this time, a nullptr is returned.

Frame data is guaranteed to be valid until `release()` has been called.

Note

This function does not remove the frame from the queue. Thus, `release()` must be called after frame data has been processed.

Note

In single counter mode, one frame is queued for each exposure. In dual counter mode, two frames with same frame number and sub frame numbers 1 and 2 are queued for each exposure. The number of sub frames can be determined by calling `numberOfSubFrames()` on the receiver object.

For discrete sensors, one frame per connected compact sensor is sent for each exposure. Each of these frames has the same frame number but different connector number. The number of connected compact sensors (i.e. chips) can be determined by calling `numberOfConnectedSensors()` on the receiver object.

Parameters

timeout_ms	timeout in milliseconds to wait for frames

Return Value

A pointer to a `Frame` object or a nullptr.

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(1);
    d->setShutterTime(0.5);
    d->startAcquisition();
    auto f = r->frame(100);
    r->release(f);

    return 0;
}
```

2.10.5. frameDepth()

```
int xsp::Receiver::frameDepth() const
```

Returns the bit depth of the frame. The value represents a number of significant bits per frame pixel. For example, a frame depth of 12 bits indicates that frame pixels are stored as 16-bit integers, where only the lower 12 bits may have non-zero values.

The frame depth is only available after the system has been initialized.

Return Value

Number of valid bits per frame pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto r = s->receiver("DET_01/1");
    std::cout << "frame depth: " << r->frameDepth() << std::endl;
```

```
    return 0;
}
```

Possible Output

```
frame depth: 12
```

2.10.6. frameHeight()

```
int xsp::Receiver::frameHeight() const
```

Returns the height of the frame.

The frame height is only available after the system has been initialized.

Return Value

Height of the frame in pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto r = s->receiver("DET_01/1");
    std::cout << "frame height: " << r->frameHeight() << std::endl;

    return 0;
}
```

Possible Output

```
frame height: 516
```

2.10.7. framesQueued()

```
int xsp::Receiver::framesQueued() const
```

Returns the number of acquired frames inside the internal frame queue. If this number is non-zero, users are able to get frames by calling the `frame()` function.

Return Value

Number of frames actually queued

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = s->receiver("DET_01/1");
```

```
    d->setFrameCount(10);
    d->setShutterTime(0.5);
    d->startAcquisition();
    usleep(10000);
    std::cout << "frames queued: " << r->framesQueued() << std::endl;

    return 0;
}
```

Possible Output

```
frames queued: 10
```

2.10.8. frameWidth()

```
int xsp::Receiver::frameWidth() const
```

Returns the width of the frame.

The frame width is only available after the system has been initialized.

Return Value

Width of the frame in pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto r = s->receiver("DET_01/1");
    std::cout << "frame width: " << r->frameWidth() << std::endl;

    return 0;
}
```

Possible Output

```
frame width: 772
```

2.10.9. id()

```
std::string xsp::Receiver::id() const
```

Returns the receiver ID.

The receiver ID is equivalent to the configured reference, detector ID plus '/' plus module number, configured as:

```
receivers:
  - ref: DET_01/1
    [...]
```

Return Value

A string with the receiver ID

Example

```
#include <iostream>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    std::cout << "receiver ID: " << r->id() << std::endl;

    return 0;
}
```

Possible Output

```
receiver ID: DET_01/1
```

2.10.10. initialize()

```
void xsp::Receiver::initialize()
```

Initializes the receiver.

This function requires that the receiver has been connected by calling `connect()` beforehand. If the receiver is disconnected, then this function throws a `RuntimeError` exception with status code `BAD_NOT_CONNECTED`.

If the communication with the detector fails, a `RuntimeError` exception with status code `BAD_COMMUNICATION_ERROR` is thrown.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01");
    r->connect();
    r->initialize();

    return 0;
}
```

2.10.11. isBusy()

```
bool xsp::Receiver::isBusy() const
```

Returns whether a receiver is busy.

A receiver is busy, if an acquisition has been started.

Return Value

True, if a receiver is busy with acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    s->connect();
```

```
    s->initialize();

    while (!r->isBusy())
        sleep(1);
    std::cout << "receiver DET_01/1 busy" << std::endl;
    while (r->isBusy())
        sleep(1);
    std::cout << "receiver DET_01/1 idle" << std::endl;

    return 0;
}
```

Output

```
receiver DET_01/1 busy
receiver DET_01/1 idle
```

2.10.12. isConnected()

```
bool xsp::Receiver::isConnected() const
```

Returns whether a receiver is connected.

A receiver is connected, if the data network connection has been established.

Return Value

True, if a receiver is connected

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    s->connect();
    if (r->isConnected())
        std::cout << "receiver DET_01/1 connected" << std::endl;

    return 0;
}
```

Possible Output

```
receiver DET_01/1 connected
```

2.10.13. isReady()

```
bool xsp::Receiver::isReady() const
```

Returns whether a receiver is ready.

A receiver is ready, if it has been successfully initialized. The receiver is then ready for data reception, i.e. `frame()` can be called to access the acquired frames.

Return Value

True, if a receiver is ready for acquisition

Example

```
#include <iostream>
#include <unistd.h>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    s->connect();
    s->initialize();

    while (!r->isReady())
        sleep(1);
    std::cout << "receiver DET_01/1 ready" << std::endl;

    return 0;
}
```

Possible Output

```
receiver DET_01/1 ready
```

2.10.14. ramAllocated()

```
bool xsp::Receiver::ramAllocated() const
```

Returns whether the RAM buffer has been allocated.

The allocation is performed in the background. Especially when configuring large buffers with several tens of GB, this may take some time (in the range of seconds). Users can call this method periodically to check, whether allocation has completed.

Note

Acquisitions can already be started before RAM has been completely allocated. It is then using only the buffer that has been allocated so far.

Return Value

True, if RAM buffer is completely allocated

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto r = s->receiver("DET_01/1");

    for (auto i = 0; i < 10; i++) {
        std::cout << "allocated: " << r->ramAllocated()
            ? "yes : "no" << std::endl;
        if (r->ramAllocated()) break;
    }

    return 0;
}
```

Possible Output

```
allocated: no
allocated: no
```

```
allocated: no
allocated: yes
```

2.10.15. release()

```
void xsp::Receiver::release(const xsp::Frame* f)
```

Releases a frame from the queue.

This function must be called after the frame data has been processed in order to remove the frame from the queue.

Parameters

f	pointer to the frame as returned by a previous call to `frame()`

Example

```
#include <iostream>
#include <libxsp.h>
#include <unistd.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = s->receiver("DET_01/1");

    d->setFrameCount(100);
    d->setShutterTime(0.5);
    d->startAcquisition();
    while (true) {
        auto f = r->frame(100);
        if (f == nullptr) break;
        // process frame data
        r->release(f);
    }

    return 0;
}
```

2.10.16. setEventHandler()

```
void xsp::Receiver::setEventHandler(
        const std::function<void(xsp::EventType,const void*)>& h
        )
```

Sets a handler, which is called on user events.

The handler is called with 2 arguments: the `EventType` and a pointer. The pointer may point to auxiliary data, depending on the event type. If no auxiliary data is required, then the pointer must be set to `nullptr`.

Parameters

h	a callable object (e.g. a lambda expression)

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    bool ready_flag = false;
    r->setEventHandler([&](auto t, const void* d) {
            switch (t) {
                case xsp::EventType::READY:
                    std::cerr << "received READY event" << std::endl;
                    ready_flag = true;
                    break;
            }
        });

    s->connect();
    s->initialize();
    while (!ready_flag)
        sleep(1);
    std::cout << "receiver ready" << std::endl;

    r->clearEventHandler();
    return 0;
}
```

Possible Output

```
received READY event
receiver ready
```

2.10.17. type()

```
string xsp::Receiver::type() const
```

Returns the receiver type.

The receiver type is equivalent to the type of the referenced detector.

Return Value

A string with the receiver type

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    std::cout << "receiver type: " << r->type() << std::endl;

    return 0;
}
```

Possible Output

```
receiver type: Lambda
```

2.10.18. userData()

```
std::string xsp::Receiver::userData(const std::string& key) const
```

Returns configured user data value associated with requested key

The method returns an empty string, if a key cannot be found.

Parameters

key	the key for a user data item

Return Value

A string representing the value of a user data key

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    std::cout << "sensor material: "
        << r->userData("sensor_material") << std::endl;

    return 0;
}
```

Possible Output

```
sensor material: Si
```

2.11. Rotation

This struct represents a rotation as three Euler angles.

Members

alpha	alpha angle
beta	beta angle
gamma	gamma angle

2.12. RuntimeError

Indicates an error while executing a detector command.

It provides the following public member methods:

`code()`	returns the numerical status code

It inherits the public member methods from `Error`.

2.12.1. code()

```
std::uint32_t xsp::RuntimeError::code() const
```

Returns the numerical value of the `StatusCode` of the executed command.

2.13. ShuffleMode

This enumeration indicates the shuffling mode used before compressing data.

Values

NO_SHUFFLE	do not shuffle data
BYTE_SHUFFLE	shuffle data on byte level
BIT_SHUFFLE	shuffle data on bit level

AUTO_SHUFFLE	shuffle on byte level for multi-byte data, otherwise do not shuffle

2.14. StatusCode

This enumeration specifies command execution status codes.

Values

GOOD	command executed successful (0x00000000)
BAD_UNEXPECTED_ERROR	an unexpected error occured (0x80010000)
BAD_INTERNAL_ERROR	an internal library error occured (0x80020000)
BAD_OUT_OF_MEMORY	failed to allocate memory (0x80030000)
BAD_RESOURCE_UNAVAILABLE	a requested resource is unavailable (0x80040000)
BAD_COMMUNICATION_ERROR	communication with detector failed (0x80050000)
BAD_DEVICE_NOT_CONNECTED	detector or receiver is not connected (0x80100000)
BAD_DEVICE_NOT_SUPPORTED	detector is not supported (0x80110000)
BAD_DEVICE_NOT_READY	detector is not ready to accept commands (0x80120000)
BAD_DEVICE_BUSY	detector is busy (0x80130000)
BAD_DEVICE_FAILURE	detector reported a failure (0x80140000)
BAD_COMMAND_NOT_ACCEPTED	command was not accepted (0x80400000)
BAD_COMMAND_NOT_IMPLEMENTED	command is not implemented (0x80410000)
BAD_COMMAND_NOT_SUPPORTED	command is not supported (0x80420000)
BAD_COMMAND_TIMED_OUT	command timed out (0x80430000)
BAD_COMMAND_FAILED	command failed (0x80440000)
BAD_ARG_OUT_OF_RANGE	command argument out of range (0x804a0000)
BAD_ARG_INVALID	command argument is invalid (0x804b0000)

2.15. System

This class represents a detector system.

It provides the following public member methods:

`connect()`	connects to a detector system
`detector()`	returns a pointer to a detector object
`detectorIds()`	returns a vector of detector IDs
`disconnect()`	disconnects a detector system
`id()`	returns the system ID
`initialize()`	initializes a detector system
`isBusy()`	returns whether a detector system is busy with acquisition
`isConnected()`	returns whether a detector system is connected
`isReady()`	returns whether a detector system is ready for acquisition
`receiver()`	returns a pointer to a receiver object
`receiverIds()`	returns a vector of receiver IDs
`reset()`	resets a detector system
`postDecoder()`	returns a pointer to a postDecoder object
`postDecoderIds()`	returns a vector of postDecoder IDs
`startAcquisition()`	starts an acquisition
`stopAcquisition()`	stops an acquisition

2.15.1. connect()

```
void xsp::System::connect()
```

Opens the control and data connection to all configured detectors.

If connections cannot be established, a `RuntimeError` exception with status code `BAD_NOT_CONNECTED` is thrown.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();

    return 0;
}
```

2.15.2. detector()

```
std::shared_ptr<xsp::Detector> xsp::System::detector(const std::string& id) const
```

Returns a pointer to the detector with the specified id.

The returned pointer needs to be casted into a pointer to a specific detector class in order to get access to detector type specific functions.

Parameters

id	detector ID

Return Value

A pointer to a `Detector` object

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    std::cout << "detector ID: " << d->id() << std::endl;
    std::cout << "detector type: " << d->type() << std::endl;
    if (d->type() == "Lambda") {
        auto l = std::dynamic_pointer_cast<xsp::lambda::Detector>(d);
        std::cout << "chip IDs: ";
        for (const auto& i: l->chipIds())
            std::cout << i << " ";
        std::cout << std::endl;
    }

    return 0;
}
```

Possible Output

```
detector ID: DET_01
detector type: Lambda
chip IDs: 103D6 103E6 103B6    103K7 103J7 103H7
```

2.15.3. detectorIds()

```
std::vector<std::string> xsp::System::detectorIds() const
```

Returns the IDs of the configured detectors.

The detector IDs are configured as:

```
detectors:
  - id: DET_01
    […]
  - id: DET_02
    […]
```

Return Value

A vector of strings with the detector IDs

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    std::cout << "detector IDs: ";
    for (const auto& i: s->detectorIds())
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Possible Output

```
detector IDs: DET_01 DET_02
```

2.15.4. disconnect()

```
void xsp::System::disconnect()
```

Closes the control and data connection to all configured detectors.

Calling this function has no effect, if the detector system is already disconnected.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->disconnect();

    return 0;
}
```

2.15.5. id()

```
std::string xsp::System::id() const
```

Returns the system ID.

The system ID is configured as:

```
system:
  id: SYSID
```

Return Value

A string with the system ID

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    std::cout << “system ID: “ << s->id() << std::endl;

    return 0;
}
```

Possible Output

```
system ID: SYS_ID
```

2.15.6. initialize()

```
void xsp::System::initialize()
```

Initializes the detector system.

This function requires that the system has been connected by calling `connect()` beforehand. If the system is disconnected, then this function throws a `RuntimeError` exception with status code `BAD_NOT_CONNECTED`.

If the communication with the detector fails, a `RuntimeError` exception with status code `BAD_COMMUNICATION_ERROR` is thrown.

Execution of this function may take some time to transfer data into the detector.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    return 0;
}
```

2.15.7. isBusy()

```
bool xsp::System::isBusy() const
```

Returns whether a detector system is busy.

A detector system is busy, if `startAcquisition()` has been called. This flag is reset, when `stopAcquisition()` is called or when all frames have been received.

Return Value

True, if a system is busy with acquisition

Example

```
#include <iostream>
```

```
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    while (!s->isReady())
        usleep(1000);

    auto d = s->detector("DET_01");
    d->start_acquisition();
    while (!s->isBusy())
        usleep(1000);
    std::cout << "system busy: " << (s->isBusy() ? "yes" : "no") << std::endl;

    return 0;
}
```

Possible Output

```
system busy: yes
```

2.15.8. isConnected()

```
bool xsp::System::isConnected() const
```

Returns whether a detector system is connected.

A detector system is connected, if `connect()` has been called and all network connections for control and data have been established.

Return Value

True, if a system is connected

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    if (s->isConnected())
        std::cout << "system connected" << std::endl;

    return 0;
}
```

Possible Output

```
system connected
```

2.15.9. isReady()

```
bool xsp::System::isReady() const
```

Returns whether a detector system is ready.

A detector system is ready, if `initialize()` has been called and all detectors and receivers were successfully initialized. The detector system is then ready for data acquisition.

If the system runs on a single host, then the system is ready, when the call to `initialize()` returns. For a detector system using multiple hosts, this is not the case, and users need to periodically check the state using this method.

Return Value

True, if a system is ready for acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    while (!s->isReady())
        sleep(1);
    std::cout << "system ready" << std::endl;

    return 0;
}
```

Possible Output

```
system ready
```

2.15.10. receiver()

```
std::shared_ptr<xsp::Receiver> xsp::System::receiver(const std::string& id) const
```

Returns a pointer to the receiver with the specified id.

Parameters

id	receiver ID

Return Value

A pointer to a `Receiver` object

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    std::cout << "receiver ID: " << r->id() << std::endl;

    return 0;
}
```

Possible Output

```
receiver ID: DET_01/1
```

2.15.11. receiverIds()

```
std::vector<std::string> xsp::System::receiverIds(const string& detector="") const
```

Returns the IDs of the configured receivers for the specified detector, or all receivers if the detector is not specified.

The receiver IDs are identical to the reference, which are configured as:

```
detectors:
  - ref: DET_01/1
    […]
  - ref: DET_01/2
    […]
```

Return Value

A vector of strings with the receiver IDs

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    std::cout << "receiver IDs: ";
    for (const auto& i: s->receiverIds())
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Possible Output

```
receiver IDs: DET_01/1 DET_01/2
```

2.15.12. reset()

```
void xsp::System::reset()
```

Resets a detector system.

This function throws a `RuntimeError` exception, if there are communication failures while writing the reset command into the detector.

Detectors are automatically reinitialized after the reset. The state of a detector after a reset is identical to the state after a system initialization.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    s->reset();

    return 0;
}
```

2.15.13. postDecoder()

```
std::shared_ptr<xsp::PostDecoder> xsp::System::postDecoder(const std::string& id) const
```

Returns a pointer to the post decoder for a specified detector.

Parameters

id	post decoder ID

Return Value

A pointer to a `PostDecoder` object

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto pd = s->postDecoder("DET_01");

    return 0;
}
```

2.15.14. postDecoderIds()

```
std::vector<std::string> xsp::System::postDecoderIds() const
```

Returns the IDs of the configured post decoders.

The post decoder IDs are configured as receiver references:

```
post-decoding:
  - ref: DET_01/1
    […]
  - ref: DET_01/2
    […]
```

Return Value

A vector of strings with the post decoder IDs

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    std::cout << "post decoder IDs: ";
    for (const auto& i: s->postDecoderIds())
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Possible Output

```
post decoder IDs: DET_01/1 DET_01/2
```

2.15.15. startAcquisition()

```
void xsp::System::startAcquisition()
```

Starts an acquisition on all detectors of a system.

This function throws a `RuntimeError` exception, if there are communication failures while writing the start command into the detectors.

The acquisition is automatically stopped, if the number of frames have been acquired, as programmed with the last call to `setFrameCount()`.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = s->detector("DET_01");
    d->setFrameCount(100);
    d->setShutterTime(0.5);

    s->startAcquisition();

    return 0;
}
```

2.15.16. stopAcquisition()

```
void xsp::System::stopAcquisition()
```

Stops a running acquisition on all detectors of a system.

This function throws a `RuntimeError` exception, if there are communication failures while writing the start command into the detectors.

Execution of this functions may take at most the time, that has been configured as link timeout (in milliseconds) for receivers.[2]

```
receivers:
  - […]
    links:
      - […]
        timeout: 1000
```

If no acquisition is running, this function has no effect and returns immediately.

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = s->detector("DET_01");
    d->setFrameCount(100);
    d->setShutterTime(1000.0);

    s->startAcquisition();
    sleep(5);
    s->stopAcquisition();

    return 0;
}
```

[2]Internally, the link timeout is used to regularly check, whether an acquisition stop has been requested.

3. Lambda Classes

The following common classes are defined in namespace `xsp::lambda`:

`BitDepth`	bit depth enumeration
`ChargeSumming`	charge summing mode enumeration
`CounterMode`	counter mode enumeration
`Detector`	Lambda detector class
`Feature`	firmware feature flag enumeration
`Gating`	gating mode enumeration
`ModuleFlag`	module flag enumeration
`OperationMode`	operation mode class
`Pitch`	pixel pitch enumeration
`Receiver`	Lambda receiver class
`Threshold`	threshold enumeration
`TrigMode`	trigger mode enumeration

3.1. BitDepth

This enumeration indicates the bit depth to use for acquisitions.

Values

DEPTH_1	1 bit depth
DEPTH_6	6 bit depth
DEPTH_12	12 bit depth
DEPTH_24	24 bit depth

3.2. ChargeSumming

This enumeration indicates whether charge summing is used.

Values

OFF	charge summing is not used
ON	charge summing is used

3.3. CounterMode

This enumeration indicates how to use the counters during acquisitions.

Values

SINGLE	only 1 counter is used during acquisition
DUAL	2 counter is used during acquisition (for 1, 6, and 12 bit depths only)

3.4. Detector

This class represents a Lambda detector.

It provides the following public member methods:

`beamEnergy()`	returns current beam energy
`bitDepth()`	returns current bit depth
`chargeSumming()`	returns current charge summing mode
`chipIds()`	returns readout chip IDs
`chipNumbers()`	returns readout chip numbers
`counterMode()`	returns current counter mode
`countrateCorrectionEnabled()`	returns whether countrate correction is enabled
`dacDisc()`	returns global DAC discriminator settings
`dacOut()`	returns output of internal test DAC
`disableCountrateCorrection()`	disables countrate correction
`disableEqualization()`	clears equalization bit in OMR
`disableFlatfield()`	disable flatfield correction
`disableInterpolation()`	disables interpolation of extra large pixels
`disableLookup()`	disables lookup of raw counter values
`disableModuleFlag()`	disables specified module flag
`disableSaturationFlag()`	disables flagging of pixel saturation
`disableTestMode()`	disables test mode
`enableCountrateCorrection()`	enables countrate correction
`enableEqualization()`	sets equalization bit in OMR
`enableFlatfield()`	enables flatfield correction
`enableInterpolation()`	enables interpolation of extra large pixels
`enableLookup()`	enables lookup of raw counter values
`enableModuleFlag()`	enables the specified module flag
`enableSaturationFlag()`	enables flagging of pixel saturation
`enableTestMode()`	enables test mode
`equalizationEnabled()`	return whether equalization bit in OMR is set
`flatfieldEnabled()`	returns whether flatfield correction is enabled
`firmwareVersion()`	returns module firmware version
`framesPerTrigger()`	returns the configured frames per trigger
`gatingMode()`	returns the current gating mode
`hasFeature()`	returns whether module firmware supports the specified feature
`humidity()`	returns measured humidity inside module
`interpolationEnabled()`	returns whether interpolation of extra large pixels is enabled
`ioDelay()`	returns the configured hardware I/O delays
`isModuleBusy()`	returns whether module is busy with acquisition
`isModuleConnected()`	returns whether module is connected
`isModuleReady()`	returns whether module is ready for acquisition
`loadTestPattern()`	loads a test pattern into a readout chip
`lookupEnabled()`	returns whether lookup of raw counter values is enabled
`moduleFlagEnabled()`	returns whether specified module flag is enabled
`numberOfModules()`	returns number of detector modules
`operationMode()`	returns current operation mode
`pixelDiscH()`	returns per pixel discH values
`pixelDiscL()`	returns per pixel discL values
`pixelMaskBit()`	returns per pixel mask bits
`pixelTestBit()`	returns per pixel test bits
`rawThresholds()`	returns raw threshold values
`readTestPattern()`	reads test pattern from a readout chip

`roiRows()`	returns the configured number of rows for ROI readout
`saturationFlagEnabled()`	returns whether flagging of saturated pixels is enabled
`saturationThreshold()`	returns the configured threshold for saturation flag
`saturationThresholdPerPixel()`	returns the configured per pixel threshold for saturation flag
`selectDiscH()`	selects discH for readout
`selectDiscL()`	selects discL for readout
`sensorCurrent()`	returns measured module sensor current
`setBeamEnergy()`	sets beam energy used for flatfield correction
`setBitDepth()`	sets bit depth
`setChargeSumming()`	sets charge summing mode
`setCounterMode()`	sets counter mode
`setDacDisc()`	sets global DAC discriminator values
`setDacIn()`	sets value for internal test DAC
`setFramesPerTrigger()`	sets number of frames to acquire per trigger
`setGatingMode()`	sets gating mode
`setIoDelay()`	sets hardware I/O delays
`setOperationMode()`	sets operation mode
`setPixelDiscH()`	sets per pixel discH values
`setPixelDiscL()`	sets per pixel discL values
`setPixelMaskBit()`	sets per pixel mask bits
`setPixelTestBit()`	sets per pixel test bits
`setRawThresholds()`	sets raw threshold values
`setRoiRows()`	sets number of rows for ROI readout
`setSaturationThreshold()`	sets threshold for saturation flag
`setSaturationThresholdPerPixel()`	sets per pixel threshold for saturation flag
`setThresholds()`	sets energy thresholds
`setTriggerMode()`	sets trigger mode
`setVoltage()`	sets value of sensor HV
`temperature()`	returns measured temperature inside module
`testModeEnabled()`	returns whether test mode is enabled
`thresholds()`	returns current energy thresholds
`triggerMode()`	returns current trigger mode
`voltage()`	returns measured sensor HV
`voltageSettled()`	returns whether sensor HV has reached configured value

It inherits all methods from `xsp::Detector` base class.

3.4.1. beamEnergy()

```
double xsp::lambda::Detector::beamEnergy() const
```

Returns the current beam energy setting in keV for the Lambda detector.

If no beam energy has been set using the setBeamEnergy() method, then 0.0 is returned.

Return Value

Beam energy in keV

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setBeamEnergy(20.0);
    std::cout << “beam energy: “ << d->beamEnergy() << std::endl;

    return 0;
}
```

Output

```
beam energy: 20.0
```

3.4.2. bitDepth()

```
xsp::lambda::BitDepth xsp::lambda::Detector::bitDepth() const
```

Returns the bit depth for the Lambda detector.

Return Value

Enumeration of type `BitDepth`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto bit_depth = d->bitDepth();
    switch (bit_depth) {
     case xsp::lambda::BitDepth::DEPTH_1:
       std::cout << "bit depth: 1" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_6:
       std::cout << "bit depth: 6" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_12:
       std::cout << "bit depth: 12" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_24:
       std::cout << "bit depth: 24" << std::endl;
       break;
    }

    return 0;
}
```

Possible Output

```
bit depth: 12
```

3.4.3. chargeSumming()

```
xsp::lambda::ChargeSumming xsp::lambda::Detector::chargeSumming() const
```

Returns the charge summing mode for the Lambda detector.

Return Value

Enumeration of type `ChargeSumming`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto cs = d->chargeSumming();
    switch (cs) {
     case xsp::lambda::ChargeSumming::OFF:
       std::cout << "charge summing: off" << std::endl;
       break;
     case xsp::lambda::ChargeSumming::ON:
       std::cout << "charge summing: on" << std::endl;
       break;
    }

    return 0;
}
```

Possible Output

```
charge summing: off
```

3.4.4. chipIds()

```
std::vector<std::string> xsp::lambda::Detector::chipIds(int module_nr) const
```

Returns a vector of strings with decoded chip IDs from a specific detector module.

This function always returns a vector of 12 strings, regardless of how many chips have been configured in the configuration file. The IDs are returned in the sequence of chip numbers, as defined in section A.1.1.

The chip ID is a 32 bit value, which is decoded into a string according to Figure 42 on page 57 of the CERN Medipix3RX Manual v1.4 or Figure 1 on page 30 of the Salland Electrical Test SPEC for Medipix 3RX. The string contains comma-separated values for the decoded chip ID, the tester, and the raw 32-bit value. The ID is the concatenation of batch and waver number (1-4095), the X coordinate (A-M), and the Y coordinate (1-11). Tester is either CRN for CERN or SAL for Salland.

This function throws a `RuntimeError` exception, if there are communication failures with the detector.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

A vector of 12 decoded chip IDs with waver number, X and Y coordinate, IDs of non-existing chips are decoded as empty strings

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto ids = d->chipIds(1);
    std::cout << “chip #1: “ << ids[0] << std::endl;

    return 0;
}
```

Possible Output

```
chip #1: w257-A01,CRN,0x00010111
```

3.4.5. chipNumbers()

```
std::vector<int> xsp::lambda::Detector::chipNumbers(int module_nr) const
```

Returns the chip numbers for the specified Lambda detector module, as defined in the `calibration.yml` file.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

Vector of chip numbers

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "chips: ";
    for (const auto &i: d->chipNumbers(1))
        std::cout << i << " ";
    std::cout << std::endl;

    return 0;
}
```

Possible Output

```
chips: 1 2 3 4 5 6 7 8 9 10 11 12
```

3.4.6. counterMode()

```
xsp::lambda::CounterMode xsp::lambda::Detector::counterMode() const
```

Returns the counter mode for the Lambda detector.

Return Value

Enumeration of type `CounterMode`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto cm = d->counterMode();
    switch (cm) {
     case xsp::lambda::CounterMode::SINGLE:
       std::cout << "counter mode: single" << std::endl;
       break;
     case xsp::lambda::CounterMode::DUAL:
       std::cout << "counter mode: dual" << std::endl;
       break;
    }

    return 0;
}
```

Possible Output

```
counter mode: single
```

3.4.7. countrateCorrectionEnabled()

```
bool xsp::lambda::Detector::countrateCorrectionEnabled() const
```

Returns whether correction of counts is enabled, either via system configuration or by a call to `enableCountrateCorrection()`.

Return Value

True, if correction of counts is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    if (d->countrateCorrectionEnabled())
        std::cout << "countrate correction: enabled" << std::endl;
    else
        std::cout << "countrate correction: disabled" << std::endl;

    return 0;
}
```

Possible Output

```
countrate correction: disabled
```

3.4.8. dacDisc()

```
std::vector<std::uint8t> xsp::lambda::Detector::dacDisc(
        int module_nr, int chip_nr
        ) const
```

Returns the values of I_DAC_DiscL and I_DAC_DiscH from the specified module and chip. Modules and chips are numbered from 1 to n.

The values are not read from hardware, but are instead cached inside the library from a previous call to `setDacDisc()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector with I_DAC_DiscL and I_DAC_DiscH value

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setDacDisc(1, 1, std::vector<std::uint8_t>{20, 50});
    auto values = d->dacDisc(1, 1);
    std::cout << “I_DAC_DiscL: “ << values[0] << std::endl;
    std::cout << “I_DAC_DiscH: “ << values[1] << std::endl;

    return 0;
}
```

Output

```
I_DAC_DiscL: 20
I_DAC_DiscH: 50
```

3.4.9. dacOut()

```
double xsp::lambda::Detector::dacOut(int module_nr, int chip_nr) const
```

Returns the measured analog DAC output from a specific detector module and chip.

Older detector firmware might not support this command. In this case, the value of 0.0 is returned.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

A measured DAC output in [V]

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "chip 1 DAC output: " << d->dacOut(1, 1) << " V" << std::endl;

    return 0;
}
```

Possible Output

```
chip 1 DAC output: 0.7 V
```

3.4.10. disableCountrateCorrection()

```
void xsp::lambda::Detector::disableCountrateCorrection() const
```

Disables correction of counts.

Whether correction has been enabled or not, can be checked on either the detector or the receiver object by calling `countrateCorrectionEnabled()`.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->disableCountrateCorrection();
    if (d->countrateCorrectionEnabled())
        std::cout << "countrate correction: enabled" << std::endl;
    else
        std::cout << "countrate correction: disabled" << std::endl;

    return 0;
}
```

Output

```
countrate correction: disabled
```

3.4.11. disableEqualization()

```
void xsp::lambda::Detector::disableEqualization()
```

Disables threshold equalization mode by clearing bit 19 of OMR register.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->disableEqualization();
    std::cout << "equalization: "
        << d->equalizationEnabled() ? "enabled" : "disabled" << std::endl;
```

```
    return 0;
}
```

Output

```
equalization: disabled
```

3.4.12. disableFlatfield()

```
void xsp::lambda::Detector::disableFlatfield() const
```

Disables flat field correction.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->disableFlatfield();
    if (d->flatfieldEnabled())
        std::cout << "flat field: enabled" << std::endl;
    else
        std::cout << "flat field: disabled" << std::endl;

    return 0;
}
```

Output

```
flat field: disabled
```

3.4.13. disableInterpolation()

```
void xsp::lambda::Detector::disableInterpolation()
```

Enables interpolation of extra large pixels at the border of each readout chip.

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->disableInterpolation();
    std::cout << "interpolation: "
        << d->interpolationEnabled() ? "enabled" : "disabled"
        << std::endl;

    return 0;
}
```

Output

```
interpolation: disabled
```

3.4.14. disableLookup()

```
void xsp::lambda::Detector::disableLookup()
```

Disables lookup of counter values, when decoding raw data. This is only useful when reading out test pattern, which have been previously loaded with `loadTestPattern()`.

This setting is only used in test mode. In normal mode the lookup is always enabled and this setting is ignored.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableTestMode();
    d->disableLookup();
    std::cout << "lookup: "
        << d->lookupEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
lookup: disabled
```

3.4.15. disableModuleFlag()

```
void xsp::lambda::Detector::disableModuleFlag(int module_nr, ModuleFLag flag)
```

Disables the specified flag for a module.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
flag	module flag

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->disableModuleFlag(1, xsp::lambda::ModuleFlag::IGNORE_ERRORS);
    std::cout << "ignore-errors: ";
    std::cout
        << (d->moduleFlagEnabled(1, xsp::lambda::ModuleFlag::IGNORE_ERRORS)
           ? "yes" : "no");
    std::cout << std::endl;

    return 0;
}
```

Output

```
ignore-errors: no
```

3.4.16. disableSaturationFlag()

```
void xsp::lambda::Detector::disableSaturationFlag() const
```

Disable flagging of pixel saturation.

Whether the flag has been enabled or not, can be checked on either the detector or the receiver object by calling `saturationFlagEnabled()`.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->disableSaturationFlag();
    if (d->saturationFlagEnabled())
        std::cout << "saturation flag: enabled" << std::endl;
    else
        std::cout << "saturation flag: disabled" << std::endl;

    return 0;
}
```

Output

```
saturation flag: disabled
```

3.4.17. disableTestMode()

```
void xsp::lambda::Detector::disableTestMode()
```

Disables test mode, data is decoded normally.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->disableTestMode();
    std::cout << "test mode: "
        << d->testModeEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
test mode: disabled
```

3.4.18. enableCountrateCorrection()

```
void xsp::lambda::Detector::enableCountrateCorrection() const
```

Enables correction of counts. Actual counts are then replaced with corrected values. The corrected values are organized as a simple lookup table, which contains corrected values for each actual count up to a maximum corrected values. All actual counts above the maximum corrected values are replaced with the maximum corrected value. The lookup table is stored in a file in the module directory. If this file does not exist, no correction is applied.

Whether correction has been enabled or not, can be checked on either the detector or the receiver object by calling `countrateCorrectionEnabled()`.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->enableCountrateCorrection();
    if (d->countrateCorrectionEnabled())
        std::cout << "countrate correction: enabled" << std::endl;
    else
        std::cout << "countrate correction: disabled" << std::endl;

    return 0;
}
```

Output

```
countrate correction: enabled
```

3.4.19. enableEqualization()

```
void xsp::lambda::Detector::enableEqualization()
```

Enables threshold equalization mode by setting bit 19 of OMR register.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableEqualization();
    std::cout << "equalization: "
        << d->equalizationEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
equalization: enabled
```

3.4.20. enableFlatfield()

```
void xsp::lambda::Detector::enableFlatfield() const
```

Enables flat field correction.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->enableFlatfield();
    if (d->flatfieldEnabled())
        std::cout << "flat field: enabled" << std::endl;
    else
        std::cout << "flat field: disabled" << std::endl;

    return 0;
}
```

Output

```
flat field: enabled
```

3.4.21. enableInterpolation()

```
void xsp::lambda::Detector::enableInterpolation()
```

Enables interpolation of extra large pixels at the border of each readout chip.

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->enableInterpolation();
    std::cout << "interpolation: "
        << d->interpolationEnabled() ? "enabled" : "disabled"
        << std::endl;

    return 0;
}
```

Output

```
interpolation: enabled
```

3.4.22. enableLookup()

```
void xsp::lambda::Detector::enableLookup()
```

Enables lookup of counter values, when decoding raw data.

This setting is only used in test mode. In normal mode the lookup is always enabled and cannot be switched off.

Example

```
#include <iostream>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableTestMode();
    d->enableLookup();
    std::cout << "lookup: "
        << d->lookupEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
lookup: enabled
```

3.4.23. enableModuleFlag()

```
void xsp::lambda::Detector::enableModuleFlag(int module_nr, ModuleFlag flag)
```

Enables the specified flag for a module.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
flag	module flag

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableModuleFlag(1, xsp::lambda::ModuleFlag::IGNORE_ERRORS);
    std::cout << "ignore-errors: ";
    std::cout
        << (d->moduleFlagEnabled(1, xsp::lambda::ModuleFlag::IGNORE_ERRORS)
           ? "yes" : "no");
    std::cout << std::endl;

    return 0;
}
```

Output

```
ignore-errors: yes
```

3.4.24. enableSaturationFlag()

```
void xsp::lambda::Detector::enableSaturationFlag() const
```

Enable flagging of pixel saturation. If the count is above a saturation threshold, measured in counts/s/pixel, then the MSB of the unused bits within the frame is set. The threshold can be set with `setSaturationThreshold()`.

Whether the flag has been enabled or not, can be checked on either the detector or the receiver object by calling `saturationFlagEnabled()`.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->enableSaturationFlag();
    if (d->saturationFlagEnabled())
        std::cout << "saturation flag: enabled" << std::endl;
    else
        std::cout << "saturation flag: disabled" << std::endl;

    return 0;
}
```

Output

```
saturation flag: enabled
```

3.4.25. enableTestMode()

```
void xsp::lambda::Detector::enableTestMode()
```

Enables test mode. In test mode the decoding is changed the following way:

- data is delivered using the chip coordinate system (first pixel is origin from chip)
- extra large pixels are not interpolated
- unconnected pixels are included (e.g. for 350K Hexa modules)
- live view is disabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableTestMode();
    std::cout << "test mode: "
        << d->testModeEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
test mode: enabled
```

3.4.26. equalizationEnabled()

```
bool xsp::lambda::Detector::equalizationEnabled() const
```

Returns whether threshold equalization mode is enabled.

Return Value

True, if equalization is enabled, false otherwise.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "equalization: "
        << d->equalizationEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
equalization: disabled
```

3.4.27. flatfieldEnabled()

```
bool xsp::lambda::Detector::flatfieldEnabled() const
```

Returns whether flat field correction has been enabled by a call to `enableFlat-field()`.

Return Value

True, if flat field correction has been enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    if (d->flatfieldEnabled())
        std::cout << "flat field: enabled" << std::endl;
    else
        std::cout << "flat field: disabled" << std::endl;

    return 0;
}
```

Possible Output

```
flat field: disabled
```

3.4.28. firmwareVersion()

```
std::string xsp::lambda::Detector::firmwareVersion(unsigned int module_nr) const
```

Returns a string with information about the firmware version, protocol versions, and feature bits.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method. The number 0 can be used to read out firmware version of a separate master board.

Parameters

module_nr	module number

Return Value

A string with the version information.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << d->firmwareVersion(1) << std::endl;

    return 0;
}
```

Possible Output

```
firmware=2.0.3 [ctrl=v0, data=v0, feat=0x0b]
```

3.4.29. framesPerTrigger()

```
int xsp::lambda::Detector::framesPerTrigger() const
```

Returns the number of frames to acquire per external trigger.

Return Value

Number of frames per trigger

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto n = d->framesPerTrigger());
    std::cout << "frames per trigger: " << n << << std::endl;

    return 0;
}
```

Output

```
frames per trigger: 1
```

3.4.30. gatingMode()

```
xsp::lambda::Gating xsp::lambda::Detector::gatingMode() const
```

Returns the gating mode for the Lambda detector.

Return Value

Enumeration of type `Gating`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto m = d->gatingMode());
    if (m == xsp::lambda::Gating::OFF)
        std::cout << "gating mode: OFF" << std::endl;

    return 0;
}
```

Output

```
gating mode: OFF
```

3.4.31. hasFeature()

```
bool xsp::lambda::Detector::hasFeature(
        int module_nr, xsp::lambda::Feature f
        ) const
```

Returns whether the specified feature is supported by the firmware.

If the module number is invalid (either 0 or larger than the number of modules), a false is returned.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
f	feature to test

Return Value

True, if the firmware feature is supported

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "features:" << std::endl;
    std::cout << "  HV adjust: ";
    std::cout << (d->hasFeature(1, xsp::lambda::Feature::FEAT_HV)
                 ? "yes" : "no");
    std::cout << std::endl;
    std::cout << "  1/6-bit  : ";
    std::cout << (d->hasFeature(1, xsp::lambda::Feature::FEAT_1_6_BIT)
```

```
                    ? "yes" : "no");
    std::cout << std::endl;

    return 0;
}
```

Possible Output

```
features:
  HV adjust: yes
  1/6-bit   : no
```

3.4.32. humidity()

```
double xsp::lambda::Detector::humidity(int module_nr) const
```

Returns the measured humidity from a specific detector module.

Older detector firmware might not support this command. In this case, the value of 0.0 is returned.

Modules are numbered from `1` to `n`, where `n` is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

A measured humidity in [%].

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "humidity: " << d->humidity(1) << "%" << std::endl;

    return 0;
}
```

Possible Output

```
humidity: 56.2%
```

3.4.33. interpolationEnabled()

```
bool xsp::lambda::Detector::interpolationEnabled() const
```

Returns whether interpolation of extra large pixels is enabled.

Return Value

True, if interpolation is enabled

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    std::cout << "interpolation: "
        << r->interpolationEnabled() ? "enabled" : "disabled"
        << std::endl;

    return 0;
}
```

Possible Output

```
interpolation: enabled
```

3.4.34. ioDelay()

```
(1) std::vector<std::uint8_t> xsp::lambda::Detector::ioDelay(
        int module_nr, int chip_nr
        )
(2) std::vector<std::uint8_t> xsp::lambda::Detector::ioDelay(
        int module_nr
        )
```

Returns the I/O delays for the specified module. This method has 2 variants:

- Variant (1) is used for generation 1 readout boards (i.e. firmware 0.0.0) and requires a chip number. It returns a vector of 9 values, where the first 8 values are the I/O delays for the 8 data lines, and the 9th value is the I/O delay for the clock line.
- Variant (2) is used for generation 2 readout boards (i.e. firmware 2.x.x). It returns a vector of 2 values, where the first value is the I/O delay for the left half of a module, and the second value is the I/O delay for the right half of the module.

Modules and chips are numbered from 1 to n.

Note

The returned values are the ones that are cached inside the library after a call to `setIoDelay()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector with delays, 9 values between 0-63 for variant (1), and 2 values between 0-31 for variant (2)

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
```

```
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    // we assume a generation 2 board
    std::vector<std::uint8_t> values(2, 0);
    values[0] = 12;
    values[1] = 13;
    d->setIoDelay(1, values);
    auto delays = d->ioDelay(1);
    std::cout << "left delay set to: "
        << (unsigned) delays[0] << std::endl;
    std::cout << "right delay set to: "
        << (unsigned) delays[1] << std::endl;

    return 0;
}
```

Output

```
left delay set to: 12
right delay set to: 13
```

3.4.35. isModuleBusy()

```
bool xsp::lambda::Detector::isModuleBusy(int module_nr) const
```

Returns whether a specific detector module is busy.

A module is busy, if `startAcquisition()` has been called. This flag is reset, when `stopAcquisition()` is called or if all frames have been acquired.

Modules are numbered from `1` to `n`, where `n` is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

True, if a detector module is busy with acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    while (!d->isModuleReady())
        usleep(1000);
    d->startAcquisition();
    std::cout << "module 1 of detector DET_01 busy: "
        << (d->isModuleBusy(1) ? "yes" : "no") << std::endl;

    return 0;
}
```

Output

```
module 1 of detector DET_01 busy: yes
```

3.4.36. isModuleConnected()

```
bool xsp::lambda::Detector::isModuleConnected(int module_nr) const
```

Returns whether a detector module is connected.

A detector module is connected, if the control network connection to readout board has been established.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method. The number 0 can be used to check connection status of a separate master board.

Parameters

module_nr	module number

Return Value

True, if a detector module is connected

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    if (d->isModuleConnected(1))
        std::cout << "module 1 of detector DET_01 connected" << std::endl;

    return 0;
}
```

Possible Output

```
module 1 of detector DET_01 connected
```

3.4.37. isModuleReady()

```
bool xsp::lambda::Detector::isModuleReady(int module_nr) const
```

Returns whether a detector module is ready.

A detector module is ready, if the module and his associated data receiver have been successfully initialized. The detector module is then ready for data acquisition.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

True, if a detector module is ready for acquisition

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    while (!d->isModuleReady(1))
        usleep(1000);
    std::cout << "module 1 of detector DET_01 ready" << std::endl;

    return 0;
}
```

Possible Output

```
module 1 of detector DET_01 ready
```

3.4.38. loadTestPattern()

```
void xsp::lambda::Detector::loadTestPattern(
        const std::vector<std::uint16_t>& pattern
        )
```

Loads the specified test pattern into all chips of all modules of the connected detector.

The pattern is a vector of 256*256 16-bit values. It is always loaded into both low and high 12-bit counters. Once loaded, the pattern can be read back with `readTestPattern()`.

This function throws a `RuntimeError` exception, if there are communication failures while writing the pattern into the detector.

Parameters

pattern	vector of 256*256 16-bit values

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::uint16_t pattern[256*256];
    for (auto i = 256*256; i-- > 0;) pattern[i] = i%4096;
    d->loadTestPattern(pattern);

    return 0;
}
```

3.4.39. lookupEnabled()

```
bool xsp::lambda::Detector::lookupEnabled() const
```

Returns whether counter value lookup is enabled or not.

Return Value

True, if lookup is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "lookup: "
        << d->lookupEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
lookup: disabled
```

3.4.40. moduleFlagEnabled()

```
bool xsp::lambda::Detector::moduleFlagEnabled(
        int module_nr, xsp::lambda::ModuleFlag flag
        ) const
```

Returns whether the specified `ModuleFlag` is enabled for a module.

If the module number is invalid (either 0 or larger than the number of modules), a false is returned.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
flag	module flag

Return Value

True if the module flag is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "ignore-errors: ";
    std::cout
        << (d->moduleFlagEnabled(1, xsp::lambda::ModuleFlag::IGNORE_ERRORS)
           ? "yes" : "no");
    std::cout << std::endl;
```

```
    return 0;
}
```

Possible Output

```
ignore-errors: no
```

3.4.41. numberOfModules()

```
int xsp::lambda::Detector::numberOfModules() const
```

Returns the number of modules for a Lambda detector.

Return Value

Number of detector modules

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "n modules: " << d->numberOfModules() << std::endl;

    return 0;
}
```

Possible Output

```
n modules: 1
```

3.4.42. operationMode()

```
xsp::lambda::OperationMode xsp::lambda::Detector::operationMode() const
```

Returns the `OperationMode` for the Lambda detector.

Return Value

Operation mode

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto om = d->operationMode();
    switch (om.bit_depth) {
     case xsp::lambda::BitDepth::DEPTH_1:
       std::cout << "bit depth: 1" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_6:
       std::cout << "bit depth: 6" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_12:
       std::cout << "bit depth: 12" << std::endl;
```

```
      break;
    case xsp::lambda::BitDepth::DEPTH_24:
      std::cout << "bit depth: 24" << std::endl;
      break;
    }

    return 0;
}
```

Possible Output

```
bit depth: 12
```

3.4.43. pixelDiscH()

```
std::vector<std::uint8t> xsp::lambda::Detector::pixelDiscH(
        int module_nr, int chip_nr
        ) const
```

Returns the values of adjustment for the higher threshold (disc_h) of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

The values are not read from hardware, but are instead cached inside the library from a previous call to `setPixelDiscH()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector with 8-bit unsigned values as 256x256 one-dimensional array in row-major order

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelDiscH(1, 1);
    auto col = 1;
    auto row = 1;
    std::cout << "disc_h at (1,1): "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Possible Output

```
disc_h at (1,1): 17
```

3.4.44. pixelDiscL()

```
std::vector<std::uint8t> xsp::lambda::Detector::pixelDiscL(
        int module_nr, int chip_nr
        ) const
```

Returns the values of adjustment for the lower threshold (disc_l) of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

The values are not read from hardware, but are instead cached inside the library from a previous call to `setPixelDiscL()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector with 8-bit unsigned values as 256x256 one-dimensional array in row-major order

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelDiscL(1, 1);
    auto col = 1;
    auto row = 1;
    std::cout << "disc_l at (1,1): "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Possible Output

```
disc_l at (1,1): 9
```

3.4.45. pixelMaskBit()

```
std::vector<std::uint8t> xsp::lambda::Detector::pixelMaskBit(
        int module_nr, int chip_nr
        ) const
```

Returns the values of the mask bit of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

The values are not read from hardware, but are instead cached inside the library from a previous call to `setPixelMaskBit()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector with 8-bit unsigned values as 256x256 one-dimensional array in row-major order

Example

```
#include <iostream>
```

```
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelMaskBit(1, 1);
    auto col = 1;
    auto row = 1;
    std::cout << "mask bit at (1,1): "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Possible Output

```
mask bit at (1,1): 0
```

3.4.46. pixelTestBit()

```
std::vector<std::uint8t> xsp::lambda::Detector::pixelTestBit(
        int module_nr, int chip_nr
        ) const
```

Returns the values of the test bit of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

The values are not read from hardware, but are instead cached inside the library from a previous call to `setPixelTestBit()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector with 8-bit unsigned values as 256x256 one-dimensional array in row-major order

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelTestBit(1, 1);
    auto col = 1;
    auto row = 1;
    std::cout << "test bit at (1,1): "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Possible Output

```
test bit at (1,1): 0
```

3.4.47. rawThresholds()

```
std::vector<unsigned> xsp::lambda::Detector::rawThresholds(
        int module_nr, int chip_nr
        ) const
```

Returns the digital 9-bit threshold values for a specific module and chip.

The thresholds are not read from hardware, but are instead cached inside the library from a previous call to `setRawThresholds()`.

Parameters

module_nr	module number
chip_nr	chip number

Return Value

Vector of eight digital threshold values as 9-bit integers

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setThresholds(std::vector<double>{6.0});
    std::cout << "raw thresholds chip #1: ",
    for (const auto& i: d->rawThresholds(1, 1)) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output

```
raw thresholds chip #1: 57 511 511 511 511 511 511 511
```

3.4.48. readTestPattern()

```
void xsp::lambda::Detector::readTestPattern()
```

Reads back the previously loaded test pattern from the detector. This basically runs a regular acquisition. To read out low and high counter, the dual counter operation mode must be selected, and the bit depth must be set to 12. In addition, the test mode must be enabled with enableTestMode(), and lookup must be disabled with disableLookup().

Important

To avoid counts due to radiation, the sensor should be covered, thresholds should be set as high as possible (i.e. 40.0 keV), and shutter time should be set to minimum (i.e. 0.5ms).

The pattern themselves are read using the frame() method from the associated receiver objects. The first received frame contains the low counter values and the second received frame contains the high counter values.

This function throws a `RuntimeError` exception, if there are communication failures while starting the read back.

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    r->enableTestMode();
    r->disableLookup();

    std::uint16_t pattern[256*256];
    for (auto i = 256*256; i-- > 0;) pattern[i] = i%4096;
    d->loadTestPattern(pattern);
    d->setFrameCount(1);
    auto om = d->operationMode();
    om.bit_depth = xsp::lambda::BitDepth::DEPTH_12;
    om.counter_mode = xsp::lambda::CounterMode::DUAL;
    d->setOperationMode(om);
    d->readTestPattern();

    auto pattern_l = r->frame(1500);
    auto pattern_h = r->frame(1500);
    // process pattern
    r->release(pattern_l);
    r->release(pattern_h);

    return 0;
}
```

3.4.49. roiRows()

```
int xsp::lambda::Detector::roiRows() const
```

Returns the numbers of readout rows per chip. A value of 256 indicates full readout, values less then 256 indicate region of interest readout.

> **Note**
>
> Region of interest readout is currently not implemented, thus this method will always return 256.

Return Value

number of readout rows per chip

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << “region of interest: “ << d->roiRows() << " rows" << std::endl;
```

```
    return 0;
}
```

Output

```
region of interest: 256 rows
```

3.4.50. saturationFlagEnabled()

```
bool xsp::lambda::Detector::saturationFlagEnabled() const
```

Returns whether flagging of pixel saturation is enabled either via system configuration or by a call to `enableSaturationFlag()`.

Return Value

True, if flagging of pixel saturation is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    if (d->saturationFlagEnabled())
        std::cout << "saturation flag: enabled" << std::endl;
    else
        std::cout << "saturation flag: disabled" << std::endl;

    return 0;
}
```

Possible Output

```
saturation flag: disabled
```

3.4.51. saturationThreshold()

```
int xsp::lambda::Detector::saturationThreshold(int module_nr) const
```

Returns actual saturation threshold in counts per second per pixel of the specified module. Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

Saturation threshold in counts per second per pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
```

```
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    std::cout << "saturation threshold: " << d->saturationThreshold(1)
        << std::endl;

    return 0;
}
```

Possible Output

```
saturation threshold: 200000
```

3.4.52. saturationThresholdPerPixel()

```
const std::vector<int> xsp::lambda::Detector::saturationThresholdPerPixel(
        int module_nr
        ) const
```

Returns the per pixel saturation thresholds in counts per second per pixel of the specified module. Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

If no per pixel saturation thresholds have been configured or set, then the returned vector has size 0.

Parameters

module_nr	module number

Return Value

Vector with per pixel saturation threshold in counts per second per pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    auto v = d->saturationThresholdPerPixel(1);
    if (v.empty())
        std::cout << "saturation threshold per pixel: not set"
            << std::endl;
    else
        std::cout << "saturation threshold per pixel: set"
            << std::endl;

    return 0;
}
```

Possible Output

```
saturation threshold per pixel: not set
```

3.4.53. selectDiscH()

```
void xsp::lambda::Detector::selectDiscH()
```

Selects output of discriminator DiscH, if threshold equalization mode is enabled with a previous call to `enableEqualization()`. If threshold equalization mode is not enabled, the method has no effect.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableEqualization();
    d->selectDiscH();
    //...

    return 0;
}
```

3.4.54. selectDiscL()

```
void xsp::lambda::Detector::selectDiscL()
```

Selects output of discriminator DiscL, if threshold equalization mode is enabled with a previous call to `enableEqualization()`. If threshold equalization mode is not enabled, the method has no effect.

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->enableEqualization();
    d->selectDiscL();
    //...

    return 0;
}
```

3.4.55. sensorCurrent()

```
double xsp::lambda::Detector::sensorCurrent(int module_nr) const
```

Returns the measured sensor current from a specific detector module.

Older detector firmware might not support this command. In this case, the value of 0.0 is returned.

Modules are numbered from `1` to `n`, where `n` is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

Measured sensor current in [mA]

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "sensor current: " << d->sensorCurrent(1) << "mA" << std::endl;

    return 0;
}
```

Possible Output

```
sensor current: 371.52mA
```

3.4.56. setBeamEnergy()

```
void xsp::lambda::Detector::setBeamEnergy(double e_kev)
```

Sets the beam energy in keV for the Lambda detector. The energy is used to select the correct flatfield.

Parameters

e_kev	beam energy in keV

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setBeamEnergy(20.0);
    std::cout << “beam energy: “ << d->beamEnergy() << std::endl;

    return 0;
}
```

Output

```
beam energy: 20.0
```

3.4.57. setBitDepth()

```
void xsp::lambda::Detector::setBitDepth(xsp::lambda::BitDepth depth)
```

Sets the specified `BitDepth` for the Lambda detector.

Older firmware does not support bit depth 1 and 6. Whether these are supported can be tested using the `hasFeature()` method with `FEAT_1_6_BIT`. If this method is called with unsupported bit depths, a `RuntimeError` exception with status code `BAD_ARG_INVALID` is thrown.

This function throws a `RuntimeError` exception, if there are communication failures while writing the value into the detector.

Parameters

depth	bit depth

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setBitDepth(xsp::lambda::BitDepth::DEPTH_24);
    auto bit_depth = d->bitDepth();
    switch (bit_depth) {
     case xsp::lambda::BitDepth::DEPTH_1:
       std::cout << "bit depth: 1" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_6:
       std::cout << "bit depth: 6" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_12:
       std::cout << "bit depth: 12" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_24:
       std::cout << "bit depth: 24" << std::endl;
       break;
    }

    return 0;
}
```

Output

```
bit depth: 24
```

3.4.58. setChargeSumming()

```
void xsp::lambda::Detector::setChargeSumming(xsp::lambda::ChargeSumming cs)
```

Sets the specified `ChargeSumming` mode for the Lambda detector.

This function throws a `RuntimeError` exception, if there are communication failures while writing the value into the detector.

Parameters

cs	charge summing mode

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setChargeSumming(xsp::lambda::ChargeSumming::ON);
    auto cs = d->chargeSumming();
    switch (cs) {
```

```
    case xsp::lambda::ChargeSumming::OFF:
      std::cout << "charge summing: off" << std::endl;
      break;
    case xsp::lambda::ChargeSumming::ON:
      std::cout << "charge summing: on" << std::endl;
      break;
    }

    return 0;
}
```

Output

```
charge suming: on
```

3.4.59. setCounterMode()

```
void xsp::lambda::Detector::setCounterMode(xsp::lambda::CounterMode cm)
```

Sets the specified `CounterMode` for the Lambda detector.

This function throws a `RuntimeError` exception, if there are communication failures while writing the value into the detector.

Parameters

cm	counter mode

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setCounterMode(xsp::lambda::CounterMode::DUAL);
    auto cm = d->counterMode();
    switch (cm) {
     case xsp::lambda::CounterMode::SINGLE:
       std::cout << "counter mode: single" << std::endl;
       break;
     case xsp::lambda::CounterMode::DUAL:
       std::cout << "counter mode: dual" << std::endl;
       break;
    }

    return 0;
}
```

Output

```
counter mode: dual
```

3.4.60. setDacDisc()

```
void xsp::lambda::Detector::setDacDisc(
        int module_nr, int chip_nr, const std::vector<std::uint8_t>& values
        )
```

Sets the values of I_DAC_DiscL and I_DAC_DiscH for the specified module and chip. Modules and chips are numbered from 1 to n.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

module_nr	module number
chip_nr	chip number
values	vector with I_DAC_DiscL and I_DAC_DiscH value

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setDacDisc(1, 1, std::vector<std::uint8_t>{20, 50});
    auto values = d->dacDisc(1, 1);
    std::cout << "I_DAC_DiscL: " << values[0] << std::endl;
    std::cout << "I_DAC_DiscH: " << values[1] << std::endl;

    return 0;
}
```

Output

```
I_DAC_DiscL: 20
I_DAC_DiscH: 50
```

3.4.61. setDacIn()

```
void xsp::lambda::Detector::setDacIn(int module_nr, int chip_nr, double voltage)
```

Sets the analog DAC input voltage for a specific detector module and chip. Voltage must be in the range of 0..1 V.

Older detector firmware might not support this command. In this case, the command has no effect.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
chip_nr	chip number

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setDacIn(1, 1, 0.78);

    return 0;
}
```

3.4.62. setFramesPerTrigger()

```
void xsp::lambda::Detector::setFramesPerTrigger(int n)
```

Sets the number of frames to acquire per external trigger, if trigger mode has been set to EXT_FRAMES.

Currently, only 1 frame per trigger is supported. If this function is called with values other than 1, a RuntimeError exception with status code BAD_ARG_OUT_OF_RANGE is thrown.

This function throws a RuntimeError exception with status code BAD_COMMUNICATION_ERROR, if there are communication failures while writing the value into the detector.

Parameters

n	frames per trigger

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setTriggerMode(xsp::lambda::TrigMode::EXT_FRAMES);
    d->setFramesPerTrigger(1);
    auto n = d->framesPerTrigger(1));
    std::cout << "frames per trigger: " << n << std::endl;

    return 0;
}
```

Output

```
frames per trigger: 1
```

3.4.63. setGatingMode()

```
void xsp::lambda::Detector::setGatingMode(xsp::lambda::Gating mode)
```

Sets the specified Gating mode for the Lambda detector.

Older firmware does not support gating in combination with trigger modes EXT_SEQUENCE and EXT_FRAMES. Whether these combinations are supported can be tested using the hasFeature() method with FEAT_EXTENDED_GATING. If this method is called resulting in an unsupported combination, a RuntimeError exception with status code BAD_ARG_INVALID is thrown.

This function throws a RuntimeError exception with status code BAD_COMMUNICATION_ERROR, if there are communication failures while writing the value into the detector.

Parameters

mode	gating mode

Example

```
#include <iostream>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setGatingMode(xsp::lambda::Gating::ON);
    auto m = d->gatingMode());
    if (m == xsp::lambda::Gating::ON)
        std::cout << "gating mode: ON" << std::endl;

    return 0;
}
```

Output

```
gating mode: ON
```

3.4.64. setIoDelay()

```
(1) void xsp::lambda::Detector::setIoDelay(
        int module_nr, int chip_nr, const std::vector<std::uint8_t>& values
        )
(2) void xsp::lambda::Detector::setIoDelay(
        int module_nr, const std::vector<std::uint8_t>& values
        )
```

Sets the I/O delays for the specified module. This method has 2 variants:

- Variant (1) is used for generation 1 readout boards (i.e. firmware 0.0.0) and requires a chip number and a vector of 9 values, where the first 8 values are the I/O delays for the 8 data lines, and the 9th value is the I/O delay for the clock line.

- Variant (2) is used for generation 2 readout boards (i.e. firmware 2.x.x) and requires a vector of 2 values, where the first value is the I/O delay for the left half of a module, and the second value is the I/O delay for the right half of the module. For 6-chip, 4-chip, and 1-chip sensor modules, only the left half value needs to be written.

Modules and chips are numbered from 1 to n. The delay values have to be between 0 and 63 for variant (1), and between 0 and 31 for variant (2). Values outside of this range are silently cropped to the maximum value.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

module_nr	module number
chip_nr	chip number
values	vector with delays, values can be 0-63 for variant (1), and 0-31 for variant (2).

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
```

```
        s->detector("DET_01"));
    // we assume a generation 2 board
    std::vector<std::uint8_t> values(2, 0);
    values[0] = 12;
    values[1] = 13;
    d->setIoDelay(1, values);
    std::cout << "left delay set to: "
        << (unsigned) values[0] << std::endl;
    std::cout << "right delay set to: "
        << (unsigned) values[1] << std::endl;

    return 0;
}
```

Output

```
left delay set to: 12
right delay set to: 13
```

3.4.65. setOperationMode()

```
void xsp::lambda::Detector::setOperationMode(
        const xsp::lambda::OperationMode& mode
        )
```

Sets the specified `OperationMode` for the Lambda detector.

Older firmware does not support bit depth 1 and 6. Whether these are supported can be tested using the `hasFeature()` method with `FEAT_1_6_BIT`. If this method is called with unsupported bit depths, a `RuntimeError` exception with status code `BAD_ARG_INVALID` is thrown.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

mode	operation mode

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setOperationMode(OperationMode(xsp::lambda::BitDepth::DEPTH_24);
    auto om = d->operationMode();
    switch (om.bit_depth) {
     case xsp::lambda::BitDepth::DEPTH_1:
       std::cout << "bit depth: 1" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_6:
       std::cout << "bit depth: 6" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_12:
       std::cout << "bit depth: 12" << std::endl;
       break;
     case xsp::lambda::BitDepth::DEPTH_24:
       std::cout << "bit depth: 24" << std::endl;
       break;
    }

    return 0;
}
```

Output

```
bit depth: 24
```

3.4.66. setPixelDiscH()

```
void xsp::lambda::Detector::setPixelDiscH(
        int module_nr, int chip_nr, const std::vector<std::uint8_t>& values
        )
```

Sets the values of the adjustment for the higher threshold (disc_h) of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

module_nr	module number
chip_nr	chip number
values	vector with 256x256 5-bit adjustments

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelDiscH(1, 1);
    auto col = 1;
    auto row = 1;
    values[row * 256 + col] = 20;
    d->setPixelDiscH(1, 1, values);
    std::cout << "disc_h at (1,1) set to: "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Output

```
disc_h at (1,1) set to: 20
```

3.4.67. setPixelDiscL()

```
void xsp::lambda::Detector::setPixelDiscL(
        int module_nr, int chip_nr, const std::vector<std::uint8_t>& values
        )
```

Sets the values of the adjustment for the lower threshold (disc_l) of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

module_nr	module number
chip_nr	chip number

values	vector with 256x256 5-bit adjustments

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelDiscL(1, 1);
    auto col = 1;
    auto row = 1;
    values[row * 256 + col] = 12;
    d->setPixelDiscL(1, 1, values);
    std::cout << "disc_l at (1,1) set to: "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Output

```
disc_l at (1,1) set to: 12
```

3.4.68. setPixelMaskBit()

```
void xsp::lambda::Detector::setPixelMaskBit(
        int module_nr, int chip_nr, const std::vector<std::uint8_t>& values
        )
```

Sets the values of the mask bit of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

module_nr	module number
chip_nr	chip number
values	vector with 256x256 mask bits

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelMaskBit(1, 1);
    auto col = 1;
    auto row = 1;
    values[row * 256 + col] = 1;
    d->setPixelMaskBit(1, 1, values);
    std::cout << "mask bit at (1,1) set to: "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Output

```
mask bit at (1,1) set to: 1
```

3.4.69. setPixelTestBit()

```
void xsp::lambda::Detector::setPixelTestBit(
        int module_nr, int chip_nr, const std::vector<std::uint8_t>& values
        )
```

Sets the values of the test bit of all pixels from the specified module and chip. Modules and chips are numbered from 1 to n.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

module_nr	module number
chip_nr	chip number
values	vector with 256x256 test bits

Example

```
#include <iostream>
#include <cstdint>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto values = d->pixelTestBit(1, 1);
    auto col = 1;
    auto row = 1;
    values[row * 256 + col] = 1;
    d->setPixelTestBit(1, 1, values);
    std::cout << "test bit at (1,1) set to: "
        << (unsigned) values[row * 256 + col] << std::endl;

    return 0;
}
```

Output

```
test bit at (1,1) set to: 1
```

3.4.70. setRawThresholds()

```
void xsp::lambda::Detector::setRawThresholds(
        int module_nr, int chip_nr, const std::vector<unsigned>& thresholds_9bit
        )
```

Sets the thresholds as 9-bit digital value for a specific module and chip. If less than 8 thresholds are specified, the remaining thresholds are not changed.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

> **Important**
>
> There are different sets of raw thresholds for single-pixel mode and for charge-summing mode. If the mode is switched, then the set of raw

thresholds is also exchanged. Therefore, raw thresholds must be written after setting the mode.

Parameters

module_nr	module number
chip_nr	chip number
thresholds_9bit	vector of up to 8 9-bit threshold values

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setRawThresholds(1, 1, std::vector<unsigned>{57});
    std::cout << “raw thresholds chip #1: “;
    for (const auto& i: d->rawThresholds(1, 1)) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output

```
raw thresholds chip #1: 57 511 511 511 511 511 511 511
```

3.4.71. setRoiRows()

```
void xsp::lambda::Detector::setRoiRows(int rows)
```

Sets the number of readout rows per chip, the number must be a power of 2. A value of 256 switches to full readout, a value of 128, 64, 32, 16, 8, 4, 2, and 1 switches to region of interest readout.

Note

Region of interest readout is currently not implemented, thus this method will only accept a value of 256.

Parameters

rows	number of readout rows per chip

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setRoiRows(256);
    std::cout << “region of interest: “ << d->roiRows() << " rows" << std::endl;

    return 0;
}
```

Output

```
region of interest: 256 rows
```

3.4.72. setSaturationThreshold()

```
void xsp::lambda::Detector::setSaturationThreshold(int module_nr, int n)
```

Sets global saturation threshold in counts per second per pixel for the specified module. Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number
n	saturation threshold in counts/s/px

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));

    d->setSaturationThreshold(1, 180000);
    std::cout << "saturation threshold: " << d->saturationThreshold(1)
        << std::endl;

    return 0;
}
```

Output

```
saturation threshold: 180000
```

3.4.73. setSaturationThresholdPerPixel()

```
void xsp::lambda::Detector::setSaturationThresholdPerPixel(
        int module_nr, const std::vector<int>& v
        )
```

Sets individual saturation thresholds in counts per second per pixel for the specified module. Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

The vector contains the saturation thresholds for each pixels of a frame, stored in row-major order.

Parameters

module_nr	module number
v	vector of per pixel saturation thresholds in counts/s/px

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
```

```
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->detector("DET_01/1"));

    vector<int> v(r->frameWidth() * r->frameHeight(), 180000);
    d->setSaturationThresholdPerPixel(1, v);

    return 0;
}
```

3.4.74. setThresholds()

```
void xsp::lambda::Detector::setThresholds(
        const std::vector<double>& thresholds_kev
        )
```

Sets the thresholds in keV for the Lambda detector. If less than 8 thresholds are specified, the remaining thresholds are not changed.

Note

If no beam energy has been set using the `setBeamEnergy()` method, then the beam energy is implicitly set to double the value of the first threshold.

This function throws a `RuntimeError` exception, if there are communication failures while writing the values into the detector.

Parameters

thresholds_kev	vector of up to 8 threshold values in keV

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setThresholds(std::vector<double>{7.0, 15.0});
    std::cout << “thresholds: “;
    for (const auto& i: d->thresholds()) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output

```
thresholds: 7.0 15.0
```

3.4.75. setTriggerMode()

```
void xsp::lambda::Detector::setTriggerMode(xsp::lambda::TrigMode mode)
```

Sets the specified `TrigMode` for the Lambda detector.

Older firmware does not support the modes `EXT_SEQUENCE` and `EXT_FRAMES` in combination with gating. Whether these combinations are supported can be tested using the hasFeature() method with FEAT_EXTENDED_GATING. If this method is called resulting in an unsupported combination, a RuntimeError exception with status code BAD_ARG_INVALID is thrown.

This function throws a RuntimeError exception with status code BAD_COMMUNICATION_ERROR, if there are communication failures while writing the value into the detector.

Parameters

mode	trigger mode

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setTriggerMode(xsp::lambda::TrigMode::SOFTWARE);
    auto m = d->triggerMode());
    if (m == xsp::lambda::TrigMode::SOFTWARE)
        std::cout << "trigger mode: SOFTWARE" << std::endl;

    return 0;
}
```

Output

```
trigger mode: SOFTWARE
```

3.4.76. setVoltage()

```
void xsp::lambda::Detector::setVoltage(int module_nr, double v)
```

Sets the high voltage for a specific detector module. Only the integer part of the specified voltage is used. The time to ramp up the voltage is approximately 7s for 200.0V and 10s for 300.0V. Newer firmware performs the ramp up in the background, so that applications need to periodically check whether voltage has reached the specified value using voltageSettled().

Important

The method may throw a BAD_COMMAND_TIMED_OUT error, if the firmware is not able to set the voltage to the desired level within the timeout as specified in the configuration file. In such a case, the system must be disconnected and connected again.

Older detector firmware might not support this command. In this case, the command throws a RuntimeError exception with status code BAD_COMMAND_NOT_SUPPORTED. Whether a firmware supports this command can be tested using the hasFeature() method with FEAT_HV.

Modules are numbered from 1 to n, where n is the value returned by the numberOfModules() method.

Parameters

module_nr	module number
v	voltage

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setVoltage(1, 200.0);
    auto u = d->voltage(1));
    std::cout << "high voltage: " << u << "V" << std::endl;

    return 0;
}
```

Possible Output

```
high voltage: 200.05V
```

3.4.77. temperature()

```
std::vector<double> xsp::lambda::Detector::temperature(int module_nr) const
```

Returns a vector of measured temperatures from a specific detector module. The number of values and their meaning depends on hardware revision of the detector readout board.

For new readout boards, three values are returned: the board temperature, the FPGA temperature, and the temperature from the humidity sensor. Older boards do not support this command, and an empty vector is returned.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

A vector of measured temperatures in [°C]

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "temperatures: ";
    for (auto t: d->temperature(1))
        std::cout << t << "° " << std::endl;
    std::cout << std::endl;

    return 0;
}
```

Possible Output

```
temperatures: 39.7° 40.2° 36.1°
```

3.4.78. testModeEnabled()

```
bool xsp::lambda::Detector::testModeEnabled() const
```

Returns whether test mode is enabled or not.

Return Value

True, if test mode is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    std::cout << "test mode: "
        << d->testModeEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
test mode: disabled
```

3.4.79. thresholds()

```
vector<double> xsp::lambda::Detector::thresholds() const
```

Returns the threshold settings in keV for the Lambda detector.

The thresholds are not read from hardware, but are instead cached inside the library from a previous call to `setThresholds()`.

Note

Initially, no thresholds in keV are set, so that a call to this method returns an empty vector.

Return Value

Vector of up to 8 threshold values in keV

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setThresholds(std::vector<double>{6.0});
    std::cout << “thresholds: “,
    for (const auto& i: d->thresholds()) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

Output

```
thresholds: 6.0
```

3.4.80. triggerMode()

```
xsp::lambda::TrigMode xsp::lambda::Detector::triggerMode() const
```

Returns the trigger mode for the Lambda detector.

Return Value

Enumeration of type `TrigMode`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto m = d->triggerMode());
    if (m == xsp::lambda::TrigMode::SOFTWARE)
        std::cout << "trigger mode: SOFTWARE" << std::endl;

    return 0;
}
```

Output

```
trigger mode: SOFTWARE
```

3.4.81. voltage()

```
double xsp::lambda::Detector::voltage(int module_nr) const
```

Returns the measured high voltage value from a specific detector module.

Older detector firmware might not support this command. In this case, a value of 0.0 is returned. Whether a firmware supports this command can be tested using the feature bits as returned by the `hasFeature()` method with `FEAT_HV`.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

The measured high voltage in [V].

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto u = d->voltage(1));
    std::cout << "high voltage: " << u << "V" << std::endl;

    return 0;
}
```

Possible Output

```
high voltage: 198.75V
```

3.4.82. voltageSettled()

```
bool xsp::lambda::Detector::voltageSettled(int module_nr) const
```

Returns whether the sensor HV for a specific detector module has reached the configured setpoint with a tolerance of 5.0V. The setpoint is either defined in the system configuration, or with a call to `setVoltage()`.

Note

Starting an acquisition before the high voltage is settled, results in incorrect counts due to sensor malfunction.

Older detector firmware might not support this command. In this case, a value of true is returned. Whether a firmware supports this command can be tested using the feature bits as returned by the `hasFeature()` method with `FEAT_HV`.

Modules are numbered from 1 to n, where n is the value returned by the `numberOfModules()` method.

Parameters

module_nr	module number

Return Value

Whether measured high voltage has reached the defined setpoint

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setVoltage(1, 200.0));
    while (!d->voltageSettled(1)) sleep(1);
    std::cout << "high voltage set" << std::endl;

    return 0;
}
```

Output

```
high voltage set
```

3.5. Feature

This enumeration indicates the features that are provided by the firmware.

Values

FEAT_HV	high voltage can be adjusted programmatically
FEAT_1_6_BIT	firmware supports 1- and 6-bit mode
FEAT_MEDIPIX_DAC_IO	Medipix DAC_OUT and EXTDAC_IN can be set and read out
FEAT_EXTENDED_GATING	gating can be combined with external trigger

3.6. Gating

This enumeration indicates whether gating is switched on or off.

Values

OFF	gating is switched off, always counting
ON	gating is switched on, counting only when gating input is low

3.7. ModuleFlag

This enumeration specifies the module flags.

Values

IGNORE	ignore module failures

3.8. OperationMode

This struct represents the Lambda operation mode.

Members

bit_depth	BitDepth	selects the frame bit depth
charge_summing	ChargeSumming	selects whether to use charge summing mode
counter_mode	CounterMode	selects whether to use single or dual counter
pitch	Pitch	selects fine pitch or spectroscopic mode

Construction

```
OperationMode()
OperationMode(BitDepth depth,
              ChargeSumming cs=ChargeSumming::OFF,
              CounterMode cm=CounterMode::SINGLE,
              Pitch p=Pitch::PITCH_55)
```

An OperationMode can be constructed using either a default constructor, or a constructor with bit depth plus optional charge summing mode, counter mode, and pitch.

3.9. Pitch

This enumeration indicates, whether single pixels or group of 4 pixels are read out.

Values

PITCH_55	fine pitch mode (55 µm pitch)

PITCH_110	spectroscopic mode (110 µm pitch)

3.10. Receiver

This class represents a Lambda receiver.

It provides the following public member methods:

`beamEnergy()`	returns current beam energy
`compressionEnabled()`	returns whether compression is enabled
`compressionLevel()`	returns current compression level
`compressor()`	returns configured compressor
`countrateCorrectionEnabled()`	returns whether countrate correction is enabled
`flatfield()`	returns current flatfield correction values
`flatfieldAuthor()`	returns the author of the actual flatfield correction values
`flatfieldEnabled()`	returns whether flatfield correction is enabled
`flatfieldError()`	returns errors for flatfield correction values
`flatfieldTimestamp()`	returns date and time of measurement of actual flatfield correction values
`interpolationEnabled()`	returns whether interpolation of extra large pixels is enabled
`lookupEnabled()`	returns whether lookup of raw counter values is enabled
`maxFrames()`	returns maximal number of frames that can be stored in RAM buffer
`numberOfConnectedSensors()`	returns number of connected sensors
`numberOfSubframes()`	returns number of subframes per exposure
`pixelMask()`	returns current pixel mask
`pixelMaskEnabled()`	returns whether pixel mask correction is enabled
`position()`	returns position of detector module
`rotation()`	returns rotation of detector module
`saturationFlagEnabled()`	returns whether flagging of saturated pixels is enabled
`saturationThreshold()`	returns current threshold for saturation flag
`saturationThresholdPerPixel()`	returns current per pixel threshold for saturation flag
`setCompressionLevel()`	sets compression level
`setShuffleMode()`	sets shuffle mode for compression
`shuffleMode()`	returns current shuffle mode for compression
`testModeEnabled()`	returns whether test mode is enabled
`thresholds()`	returns current energy thresholds

It inherits all methods from `xsp::Receiver` base class.

3.10.1. beamEnergy()

```
double xsp::lambda::Receiver::beamEnergy() const
```

Returns the currently set beam energy.

If energy or threshold has not been set using either `setBeamEnergy()` or `set-Thresholds() on the detector object, then 0.0 is returned.

Return Value

The actual set beam energy

Example

```
#include <iostream>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    d->setBeamEnergy(20.0);
    auto e = r->beamEnergy();
    std::cout << "beam energy: " << e << std::endl;

    return 0;
}
```

Possible Output

```
beam energy: 20.0
```

3.10.2. compressionEnabled()

```
bool xsp::lambda::Receiver::compressionEnabled() const
```

Returns whether compression is enabled. The compression is enabled via system configuration file by setting the compressor to something else than "none".

Return Value

True, if compression is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto r = s->receiver("DET_01/1");
    std::cout << "compression: "
        << r->compressionEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Possible Output

```
compression: enabled
```

3.10.3. compressionLevel()

```
int xsp::lambda::Receiver::compressionLevel() const
```

Returns the configured compression level.

Return Value

Compression level

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto r = s->receiver("DET_01/1");
    std::cout << "compression level: " << r->compressionLevel() << std::endl;

    return 0;
}
```

Possible Output

```
compression level: 4
```

3.10.4. compressor()

```
std::string xsp::lambda::Receiver::compressor() const
```

Returns the configured compressor. If no compressor has been configured, then the string "none" is returned.

Return Value

A string with the configured compressor

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto r = s->receiver("DET_01/1");
    std::cout << "compressor: " << r->compressor() << std::endl;

    return 0;
}
```

Possible Output

```
compressor: zlib
```

3.10.5. countrateCorrectionEnabled()

```
bool xsp::lambda::Receiver::countrateCorrectionEnabled() const
```

Returns whether correction of counts is enabled either via system configuration or by a call to `enableCountrateCorrection()`.

Return Value

True, if correction of counts is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
```

```
    if (r->countrateCorrectionEnabled())
        std::cout << "countrate correction: enabled" << std::endl;
    else
        std::cout << "countrate correction: disabled" << std::endl;

    return 0;
}
```

Possible Output

```
countrate correction: disabled
```

3.10.6. flatfield()

```
const std::vector<double>& xsp::lambda::Receiver::flatfield(
        xsp::lambda::Threshold th
        ) const
```

Returns a reference to the flat field correction data for the specified `Threshold`.

If no flat field file has been configured in the system configuration, then the vector is empty.

Parameters

th	threshold (LOWER or UPPER)

Return Value

A reference to a vector of double precision floating points

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    auto ff = r->flatfield(xsp::lambda::Threshold::LOWER);
    if (ff.empty())
        std::cout << "flat field: not defined" << std::endl;
    else
        std::cout << "flat field: defined" << std::endl;

    return 0;
}
```

Possible Output

```
flat field: defined
```

3.10.7. flatfieldAuthor()

```
std::string xsp::lambda::Receiver::flatfieldAuthor(
        xsp::lambda::Threshold th) const
```

Returns the author of the currently used flat field for the specified `Threshold`.

If no flat field is defined, i.e. the flatfield() method would return a zero size vector, an empty string is returned.

Parameters

th	threshold (LOWER or UPPER)

Return Value

String naming the flat field author

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    auto ff_author = r->flatfieldAuthor(xsp::lambda::Threshold::LOWER);
    std::cout << "flat field author: " << ff_author << std::endl;

    return 0;
}
```

Possible Output

```
flat field author: X-Spectrum
```

3.10.8. flatfieldEnabled()

```
bool xsp::lambda::Receiver::flatfieldEnabled() const
```

Returns whether flat field correction has been enabled either via configuration or by a call to `enableFlatfield()`.

Return Value

True, if flat field correction is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    if (r->flatfieldEnabled())
        std::cout << "flat field: enabled" << std::endl;
    else
        std::cout << "flat field: disabled" << std::endl;

    return 0;
}
```

Possible Output

```
flat field: disabled
```

3.10.9. flatfieldError()

```
const std::vector<double>& xsp::lambda::Receiver::flatfieldError(
        xsp::lambda::Threshold th
```

```
        ) const
```

Returns a reference to the error in the flat field correction data for the specified <<ref-lmb-threshold,Threshold>.

If no flat field file has been configured in the system configuration, then the vector is empty.

Parameters

th	threshold (LOWER or UPPER)

Return Value

A reference to a vector of floating points

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    auto ff_error = r->flatfieldError(xsp::lambda::Threshold::LOWER);
    if (ff_error.empty())
        std::cout << "flat field error: not defined" << std::endl;
    else
        std::cout << "flat field error: defined" << std::endl;

    return 0;
}
```

Possible Output

```
flat field error: not defined
```

3.10.10. flatfieldTimestamp()

```
std::string xsp::lambda::Receiver::flatfieldTimestamp(
        xsp::lambda::Threshold th) const
```

Returns the timestamp of the currently used flat field for the specified `Threshold`.

If no flat field is defined, i.e. the `flatfield()` method returns a zero size vector, then an empty string is returned.

Parameters

th	threshold (LOWER or UPPER)

Return Value

A date time string

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
```

```
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    auto ff_timestamp = r->flatfieldTimestamp(xsp::lambda::Threshold::LOWER);
    std::cout << "flat field timestamp: " << ff_timestamp << std::endl;

    return 0;
}
```

Possible Output

```
flat field timestamp: Tue, 30 Mar 2021 09:13:41 +0200
```

3.10.11. interpolationEnabled()

```
bool xsp::lambda::Receiver::interpolationEnabled() const
```

Returns whether interpolation of extra large pixels is enabled.

Return Value

True, if interpolation is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    std::cout << "interpolation: "
        << r->interpolationEnabled() ? "enabled" : "disabled"
        << std::endl;

    return 0;
}
```

Possible Output

```
interpolation: enabled
```

3.10.12. lookupEnabled()

```
bool xsp::lambda::Receiver::lookupEnabled() const
```

Returns whether counter value lookup is enabled or not.

Return Value

True, if lookup is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
    std::cout << "lookup: "
        << r->lookupEnabled() ? "enabled" : "disabled" << std::endl;
```

```
    return 0;
}
```

Output

```
lookup: disabled
```

3.10.13. maxFrames()

```
int xsp::lambda::Receiver::maxFrames() const
```

Returns the number of frames that can be stored in the RAM buffer.

The number of frames is only available after the system has been initialized. It may change, if the bit depth is changed.

Return Value

Size of RAM buffer in number of frames

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    s->initialize();
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    std::cout << "max frames: " << r->maxFrames() << std::endl;

    return 0;
}
```

Possible Output

```
max frames: 99871
```

3.10.14. numberOfConnectedSensors()

```
int xsp::lambda::Receiver::numberOfConnectedSensors() const
```

Returns the number of connected sensors in case of Hydra type Lambda detectors with discrete compact sensors. For normal detectors, such as 60K, 250K, 350K, 750K, a value of 1 is returned.

Return Value

Number of connected sensors

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
    std::cout << "connected sensors: "
```

```
        << r->numberOfConnectedSensors() << std::endl;

    return 0;
}
```

Possible Output

```
connected sensors: 1
```

3.10.15. numberOfSubFrames()

```
int xsp::lambda::Receiver::numberOfSubFrames() const
```

Returns the number of subframes per exposure. This number is usually 1. For dual counter mode, 2 subframes are acquired per exposure.

For discrete sensor layout, the number of subframes is equal to the number of connected single chip sensors. If in addition dual counter mode is enabled, then twice the number of connected chip sensors is returned.

When reading the decoded frames from the receiver, all subframes will have the same frame number, but different subframe numbers. Subframe number always starts at 1.

Return Value

Number of subframes per exposure

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
    std::cout << "n subframes: "
        << r->numberOfSubFrames() << std::endl;

    return 0;
}
```

Possible Output

```
n subframes: 1
```

3.10.16. pixelMask()

```
const vector<uint32_t>& xsp::lambda::Receiver::pixelMask() const
```

Returns a reference to the pixel mask, which is a vector of 32-bit integers with values defined by the NeXus standard. The values are stored in row- major order.

If no pixel mask file has been configured in the system configuration, then the vector is empty.

Return Value

A reference to a vector of 32-bit unsigned integers

Example

```
#include <iostream>
```

```
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
    auto mask = r->pixelMask();
    if (mask.empty())
        std::cout << "pixel mask: not defined" << std::endl;
    else
        std::cout << "pixel mask: defined" << std::endl;

    return 0;
}
```

Possible Output

```
pixel mask: defined
```

3.10.17. pixelMaskEnabled()

```
bool xsp::lambda::Receiver::pixelMaskEnabled() const
```

Returns whether processing of pixel mask has been enabled.

Return Value

True, if pixel mask processing is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
    if (r->pixelMaskEnabled())
        std::cout << "pixel mask: enabled" << std::endl;
    else
        std::cout << "pixel mask: not enabled" << std::endl;

    return 0;
}
```

Possible Output

```
pixel mask: not enabled
```

3.10.18. position()

```
xsp::Position xsp::lambda::Receiver::position() const
```

Returns the `Position` of the associated detector module.

Return Value

A position

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    auto pos = r->position();
    std::cout << "module position: ("
        << pos.x << "," << pos.y << "," << pos.z << ")" << std::endl;

    return 0;
}
```

Possible Output

```
module position: (0,0,0)
```

3.10.19. rotation()

```
xsp::Rotation xsp::lambda::Receiver::rotation() const
```

Returns the `Rotation` of the associated detector module.

Return Value

A rotation

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = s->receiver("DET_01/1");
    auto rot = r->rotation();
    std::cout << "module rotation: ("
        << rot.alpha << "," << rot.beta << "," << rot.gamma << ")" << std::endl;

    return 0;
}
```

Possible Output

```
module rotation: (0,0,0)
```

3.10.20. saturationFlagEnabled()

```
bool xsp::lambda::Receiver::saturationFlagEnabled() const
```

Returns whether flagging of pixel saturation is enabled or not

Return Value

True, if flagging of saturated pixels is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
```

```
    if (r->saturationFlagEnabled())
        std::cout << "saturation flag: enabled" << std::endl;
    else
        std::cout << "saturation flag: disabled" << std::endl;

    return 0;
}
```

Possible Output

```
saturation flag: disabled
```

3.10.21. saturationThreshold()

```
int xsp::lambda::Receiver::saturationThreshold() const
```

Returns actual saturation threshold in counts per second per pixel.

Return Value

Saturation threshold in counts per second per pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    std::cout << "saturation threshold: " << r->saturationThreshold()
        << std::endl;

    return 0;
}
```

Possible Output

```
saturation threshold: 200000
```

3.10.22. saturationThresholdPerPixel()

```
const std::vector<int>& xsp::lambda::Receiver::saturationThresholdPerPixel(
        ) const
```

Returns the per pixel saturation thresholds in counts per second per pixel.

If no per pixel saturation thresholds have been configured or set, then the returned vector has size 0.

Return Value

Vector with per pixel saturation threshold in counts per second per pixel

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
```

```
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));

    auto v = r->saturationThresholdPerPixel();
    if (v.empty())
        std::cout << "saturation threshold per pixel: not set"
            << std::endl;
    else
        std::cout << "saturation threshold per pixel: set"
            << std::endl;

    return 0;
}
```

Possible Output

```
saturation threshold per pixel: not set
```

3.10.23. setCompressionLevel()

```
void xsp::lambda::Receiver::setCompressionLevel(int level)
```

Sets the compression level.

The level must be in the range of 0 to 9. A value of 0 means no compression, a value of 1 means fastest compression, and a value of 9 means best compression.

Parameters

level	compression level

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = s->receiver("DET_01/1");

    r->setCompressionLevel(6);
    std::cout << "compression level: " << r->compressionLevel() << std::endl;

    return 0;
}
```

Possible Output

```
compression level: 6
```

3.10.24. setShuffleMode()

```
void xsp::lambda::Receiver::setShuffleMode(xsp::ShuffleMode mode)
```

Sets the specified `ShuffleMode` used during compression.

Parameters

mode	shuffle mode

Example

```
#include <iostream>
#include <unistd.h>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");

    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    auto r = s->receiver("DET_01/1");

    r->setShuffleMode(xsp::ShuffleMode::BYTE_SHUFFLE);
    switch (r->shuffleMode()) {
     case xsp::ShuffleMode::NO_SHUFFLE:
       std::cout << "shuffle mode: NO_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BYTE_SHUFFLE:
       std::cout << "shuffle mode: BYTE_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BIT_SHUFFLE:
       std::cout << "shuffle mode: BIT_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::AUTO_SHUFFLE:
       std::cout << "shuffle mode: AUTO_SHUFFLE" << std::endl;
       break;
    }

    return 0;
}
```

Output

```
shuffle mode: BYTE_SHUFFLE
```

3.10.25. shuffleMode()

```
xsp::ShuffleMode xsp::lambda::Receiver::shuffleMode() const
```

Returns the configured `ShuffleMode`.

Return Value

Enumeration of type `ShuffleMode`

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    s->connect();
    auto r = s->receiver("DET_01/1");
    switch (r->shuffleMode()) {
     case xsp::ShuffleMode::NO_SHUFFLE:
       std::cout << "shuffle mode: NO_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BYTE_SHUFFLE:
       std::cout << "shuffle mode: BYTE_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::BIT_SHUFFLE:
       std::cout << "shuffle mode: BIT_SHUFFLE" << std::endl;
       break;
     case xsp::ShuffleMode::AUTO_SHUFFLE:
       std::cout << "shuffle mode: AUTO_SHUFFLE" << std::endl;
```

```
        break;
    }

    return 0;
}
```

Possible Output

```
shuffle mode: AUTO_SHUFFLE
```

3.10.26. testModeEnabled()

```
bool xsp::lambda::Receiver::testModeEnabled() const
```

Returns whether test mode is enabled.

Return Value

True, if test mode is enabled

Example

```
#include <iostream>
#include <libxsp.h>

int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto r = std::dynamic_pointer_cast<xsp::lambda::Receiver>(
        s->receiver("DET_01/1"));
    std::cout << "test mode: "
        << r->testModeEnabled() ? "enabled" : "disabled" << std::endl;

    return 0;
}
```

Output

```
test mode: disabled
```

3.10.27. threshold()

```
double xsp::lambda::Receiver::threshold(xsp::lambda::Threshold th) const
```

Returns the currently set value of the specified `Threshold`.

If no thresholds have been set using `setThresholds()` on the detector object, then a value of 0.0 is returned.

Parameters

`

th	threshold (LOWER or UPPER)

Return Value

Value of specified threshold

Example

```
#include <iostream>
#include <libxsp.h>
```

```
int main()
{
    auto s = xsp::createSystem("/path/to/system.yml");
    auto d = std::dynamic_pointer_cast<xsp::lambda::Detector>(
        s->detector("DET_01"));
    d->setThresholds(std::vector<double>{10.0});
    auto r = s->receiver("DET_01/1");
    auto th = r->threshold(xsp::lambda::Threshold::LOWER);
    std::cout << "threshold: " << th << std::endl;

    return 0;
}
```

Possible Output

```
threshold: 10.0
```

3.11. Threshold

This enumeration indicates the threshold to select.

Values

LOWER	select the lower threshold
UPPER	select the upper threshold

3.12. TrigMode

This enumeration indicates the trigger mode to use for acquisitions.

Values

SOFTWARE	acquisition is started after `startAcquisition()` is called
EXT_SEQUENCE	rising edge on trigger input starts acquisition of all frames
EXT_FRAMES	for 24 bit depth, a rising edge on trigger input starts acquisition of one or more frames with the programmed shutter time, trigger input is ignored until counter has been read out; for all other bit depths, a rising edge on trigger input stops acquisition of current frame and immediately starts acquisition of next frame, the programmed shutter time is ignored in this case

Index

S

T

U

V

W