libxsp

User Manual

Version 2.1

Date 2022-05-17

libxsp: User Manual

X-Spectrum GmbH

Version 2.1
Date 2022-05-17

Table of Contents

List of Figures

List of Tables

Preface

This manual contains information that users need to build their own detector applications. It describes the programming interface of the low-level library.

Use of this software is bound by the License Agreement included with the library.

Related documents

The following related documents are available from X-Spectrum

- liblambda Migration Guide
- API Reference
- Detector User Manual
- Server Tuning Guide

Support

Support for the software library is available through your X-Spectrum sales representative.

1. Introduction

libxsp is a software library for developing detector applications. It supports the following detectors:

- Lambda 60K/250K as of firmware version 2.0.0
- Lambda 350K/750K
- multi-module Lambda 1.5M/2M/9M

Main features of the library are:

- connect to a detector system
- set acquisition parameters
- run acquisitions
- read out acquired frames
- provide live frames

This manual describes general concepts, design and use of the software library.

1.1. General Concepts

A detector system consists of detectors and hosts, which are connected with Ethernet (IEEE 802.3) networks. A detector may consist of one or more modules to increase sensor size. Control and data are typically exchanged over separate networks.

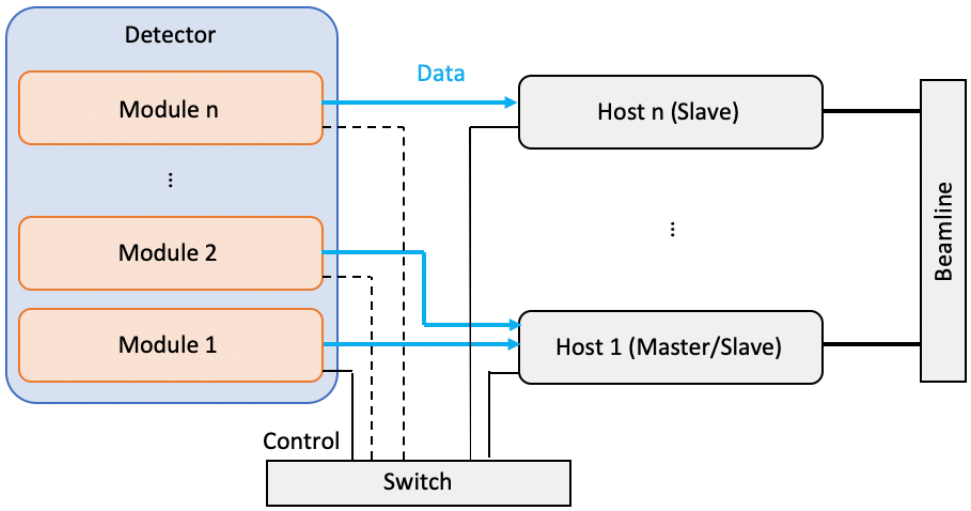

Figure 1.1. Detector System Overview

As shown in Figure 1.1, the control network connects all hosts and at least one module from the detector. For some detectors, the control network may also connect to all modules. The data network is a point to point connection between one detector module and one host, or between several modules and one host.

Hosts can have one of two roles: master or slave. The master is in authority for detector control, and slaves are in authority for data reception. A host can also have both roles (as shown with host 1). On each host, an instance of the detector

library is running, e.g. as part of a device server for the TANGO or EPICS control system. The instances communicate with each other over the control network in order to synchronize themselves.

1.2. Library Architecture

The low-level library has a layered architecture as shown in Figure 1.2.

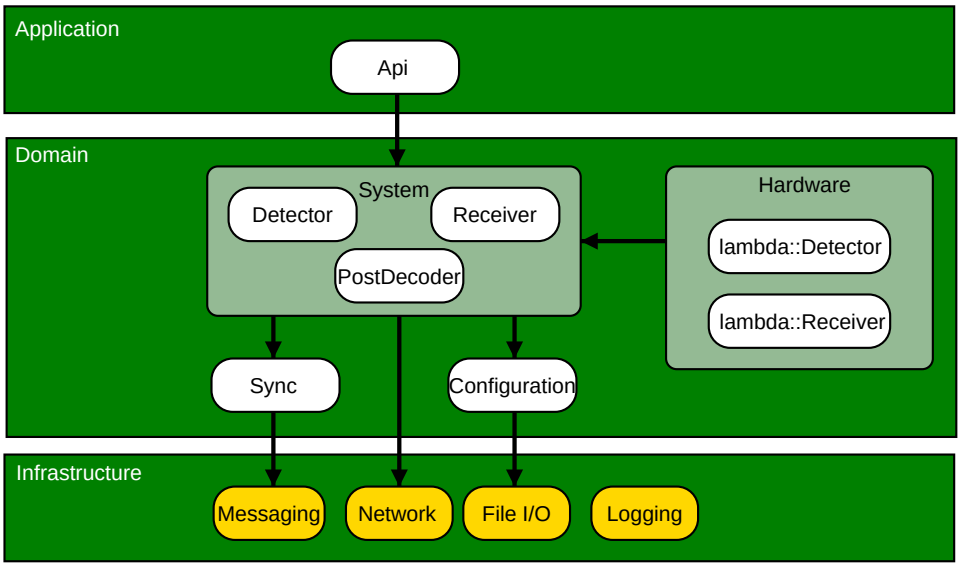

Figure 1.2. Architecture of low-level Library

The *Api* in the application layer provides the interface for the users. Access is implemented by a set of interfaces (C++ pure virtual classes).

The core logic is implemented in the domain layer. A *Detector* converts high level user commands into messages and sends them via network to the detector firmware. A *Receiver* decodes the raw data from the detector into meaningful images and presents them to the user. An optional *PostDecoder* allows to sum image data on the fly and to stitch images from multiple modules into a single image. *Detector*, *Receiver* and *PostDecoder* are configured from information contained in files.

The hardware specific logic is put into separate objects: one for detectors, one for receivers. Currently, only *LambdaDetector* and *LambdaReceiver* are supported. Future extensions of the library will add more hardware objects to support more detector types.

The *Sync* block synchronizes information between instances of *Systems* running on separate hosts. It shares configuration of *Detectors* and *Receivers* across all involved hosts, and also exchanges messages to indicate important system events such as begin and end of acquisitions.

The infrastructure layer contains objects to interface to the network and the filesystem.

1.3. System Requirements

Operating System

The X-Spectrum detector library has been tested on the following operating systems:

- Debian 9 and 10

Dependencies

The library requires additional libraries to be installed:

- libyaml-cpp
- libczmq
- libmsgpack
- libnuma
- zlib
- libblosc

The exact Debian package names usually have a version number attached, which is depending on the installed OS version. The version number is not shown in the above list, before installing one of these packages, one need to run a command like

```
apt-cache search zlib
```

to find out the exact package name.

1.4. Compilation

The library is written using C++14 standard. It makes use of std::thread, so that programs need to be linked against a thread library. On GNU/Linux, this is usually pthread. The command to compile an application using the low-level library libxsp is

```
c++ -std=c++14 application.cpp -lxsp -pthread
```

The C++14 standard is fully supported since gcc 6.1 or clang 3.4.

2. Basic Examples

This section contains example code for running simple acquisition in case of single or multi module Lambda detectors.

2.1. Single Module Detector

Let's start with a simple detector system, consisting of one Lambda detector of type 750K, which has a control and data link to a single host.

2.1.1. Configuration

The configuration file on that host, usually stored as `/etc/opt/xsp/system.yml`, would look like

```
detectors:
  - id: lambda
    type: Lambda
    modules:
      - directory: Module_1
        control:
          ip: 169.254.1.2

receivers:
  - ref: lambda/1
    type: Lambda
    links:
      - ip: 169.254.2.1
        mac: a0:b3:fd:01:ff:00
      - ip: 169.254.3.1
        mac: a0:b3:fd:01:ff:01
```

It defines one detector of type `Lambda` at the IP address 169.254.1.2 and one receiver of type `Lambda` with two links at IP addresses 169.254.2.1 and 169.254.3.1. Please note that the data link IP addresses are the IP addresses on the server side, not on the detector side. It is important, that the MAC addresses are specified correctly, otherwise data packets from the detector are discarded. To find out IP and MAC address of an interface, one can run the

```
ip a
```

command.

A full description of all configurable parameters is provided in Section 2.1.1 and in the appendix for the specific detector type.

2.1.2. Program Code

An example program to run an acquisition of 100 frames with shutter time of 0.5 ms is shown below:

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

using namespace xsp;

int main()
{
    // step 1: create system
    auto s = createSystem("/etc/opt/xsp/system.yml");
    if (s == nullptr) {
        std::cout << "No system created. Aborting..." << std::endl;
        return -1;
```

```
    }
    try {
        s->connect();
        s->initialize();
    }
    catch(const RuntimeError& e) {
        std::cout << "exception: " << e.what() << std::endl;
        return -1;
    }

    // step 2: get pointers and wait for RAM buffer
    auto d = s->detector("lambda");
    auto r = s->receiver("lambda/1");
    while (!r->ramAllocated()) sleep(1);

    // step 3: set parameters
    try {
        d->setFrameCount(100);
        d->setShutterTime(0.5);
    }
    catch(const RuntimeError& e) {
        std::cout << "exception: " << e.what() << std::endl;
        return -1;
    }

    // step 4: run acquisition
    // Note: for Lambda detectors, applications need to wait until sensor HV
    //       is set using the following loop:
    //          ld = std::dynamic_pointer_cast<lambda::Detector>(d);
    //          while (!ld->voltageSettled()) sleep(1);
    try {
        d->startAcquisition();
    }
    catch(const RuntimeError& e) {
        std::cout << "exception: " << e.what() << std::endl;
        return -1;
    }
    auto n_received = 0;
    while (true) {
        auto f = r->frame(1500);
        if (f != nullptr) {
            auto dp = reinterpret_cast<const std::uint16_t*>(f->data());
            auto size = f->size();
            //process data: e.g. std::memcpy(<dest>, dp, size);
            r->release(f);
            n_received++;
        } else {
            // timeout occured, when reading frames
            break;
        }
        if (n_received == 100)
            break;
    }

    // step 5: clean up
    s->disconnect();
    return 0;
}
```

The steps to perform an acquisition are:

1. create a `System` by calling `createSystem()` with the name of the configuration file. If a nullptr is returned, something went wrong (e.g. configuration file was not readable) and the program exits. Otherwise the system is connected and initialized. The `initialize()` call may take some time to execute, since calibration data are downloaded to the detector. These calls may throw exceptions, if communication to the detector fails.

2. get pointers to the `Detector` and `Receiver` objects by calling `detector()` and `receiver()` with the respective IDs. Since RAM buffer for intermediate

storage of raw data from the detector is allocated in the background, applications need to wait until RAM buffer is completely allocated.

3. set acquisition parameters. These calls may throw exceptions, if communication to the detector fails.

4. run acquisition and read out frames in a loop. The `frame()` method takes a timeout in milliseconds as an argument. If the timeout expires before a frame can be read out, a nullptr is returned. The data pointer needs to be type casted to the appropriate type. In this case, the assumption is that the detector delivers 12-bit data, so that the data is an array of 16-bit unsigned integers.

5. clean up and exit. `disconnect()` is called implicitly when the program terminates, thus can safely be left out.

2.2. Multi Module Detector

Now let's turn to a more complicated detector system, consisting of one Lambda detector of type 2M with 3 vertically stacked modules, which has control and data links to a single host.

2.2.1. Configuration

The configuration file on that host, usually stored as `/etc/opt/xsp/system.yml`, would look like

```
detectors:
  - id: lambda
    type: Lambda
    modules:
      - directory: Module_1
        position: { x: 0.0, y: 0.0, z: 0,0 }
        control:
          ip: 169.254.1.18
      - directory: Module_2
        position: { x: 0.0, y: 647.0, z: 0,0 }
        control:
          ip: 169.254.1.10
      - directory: Module_3
        position: { x: 0.0, y: 1294.0, z: 0,0 }
        control:
          ip: 169.254.1.2

receivers:
  - ref: lambda/1
    type: Lambda
    links:
      - ip: 169.254.2.1
        mac: a0:b3:fd:01:ff:00
      - ip: 169.254.3.1
        mac: a0:b3:fd:01:ff:01
  - ref: lambda/2
    type: Lambda
    links:
      - ip: 169.254.4.1
        mac: a0:b3:fd:7a:03:00
      - ip: 169.254.5.1
        mac: a0:b3:fd:7a:03:01
  - ref: lambda/3
    type: Lambda
    links:
      - ip: 169.254.6.1
        mac: a0:b3:fd:ed:a1:00
      - ip: 169.254.7.1
        mac: a0:b3:fd:ed:a1:01
```

It defines one detector of type `Lambda` and three receivers of type `Lambda`, each with two links. Each module has its own module directory, which contains the calibration files for that module. The module directories are specified relative to the directory containing the system configuration file. The positions of each module are defined as coordinates, with the origin being at the top left corner of the detector.

A full description of all configurable parameters is provided in Section 2.1.1 and in the appendix for the specific detector type.

Additionally, a post decoder can be used to stitch the frames from multiple modules into a single frame using integer position correction. Therefore, the config file needs to specify an additional `postdecoders:` section.

```
postdecoders:
  - ref: lambda
```

2.2.2. Program Code

An example program to run an acquisition of 100 frames with shutter time of 0.5 ms is shown below:

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

using namespace xsp;

int main()
{
    // step 1: create system
    auto s = createSystem("/etc/opt/xsp/system.yml");
    if (s == nullptr) {
        std::cout << "No system created. Aborting..." << std::endl;
        return -1;
    }
    try {
        s->connect();
        s->initialize();
    }
    catch(const RuntimeError& e) {
        std::cout << "exception: " << e.what() << std::endl;
        return -1;
    }

    // step 2: get pointers and wait for RAM buffer
    auto d = s->detector("lambda");
    auto r1 = s->receiver("lambda/1");
    auto r2 = s->receiver("lambda/2");
    auto r3 = s->receiver("lambda/3");
    while (!r1->ramAllocated()) sleep(1);
    while (!r2->ramAllocated()) sleep(1);
    while (!r3->ramAllocated()) sleep(1);

    // step 3: set parameters
    try {
        d->setFrameCount(100);
        d->setShutterTime(0.5);
    }
    catch(const RuntimeError& e) {
        std::cout << "exception: " << e.what() << std::endl;
        return -1;
    }

    // step 4: run acquisition
    // Note: for Lambda detectors, applications need to wait until sensor HV
    //       is set using the following loop:
    //          ld = std::dynamic_pointer_cast<lambda::Detector>(d);
```

```
    //          while (!ld->voltageSettled()) sleep(1);
    try {
        d->startAcquisition();
    }
    catch(const RuntimeError& e) {
        std::cout << "exception: " << e.what() << std::endl;
        return -1;
    }
    auto n_received = 0;
    while (true) {
        auto f1 = r1->frame(1500);
        auto f2 = r2->frame(1500);
        auto f3 = r3->frame(1500);
        if (f1 != nullptr) {
            // handle module 1 frame
            r1->release(f1);
            n_received++;
        }
        if (f2 != nullptr) {
            // handle module 2 frame
            r2->release(f2);
        }
        if (f3 != nullptr) {
            // handle module 3 frame
            r3->release(f3);
        }

        if (n_received == 100)
            break;
    }

    // step 5: clean up
    s->disconnect();
    return 0;
}
```

The steps to perform an acquisition are the same as in the single module example. However, receiving the frames differs, since now the frames have to be read from three different receivers. The example code is rather simple by reading the frames sequentially from each receiver. A more sophisticated way would be to use separate threads to handle the readout in parallel.

If using a post decoder, the code need to be adapted in the following way.

```
int main()
{
    [...]
    auto p = s->postDecoder("lambda");
    auto n_received = 0;
    while (true) {
        auto f = p->frame(1500);
        if (f != nullptr) {
            // handle frame
            p->release(f);
            n_received++;
        }
        if (n_received == 100)
            break;
    }
    [...]
}
```

3. Programming Interface

The Application Programming Interface (API) is a set of abstract interfaces to the objects, as described in Section 1.2: `System`, `Detector`, `Receiver`, `PostDecoder`, and the hardware specific `lambda::Detector` and `lambda::Receiver`.

This section gives an overview of the general API methods, valid for all types of detector systems, grouped by functionality. The hardware specific methods are described in the appendix.

A full description of all objects and methods is given in [2].

3.1. Initialization

In order to run acquisitions, the detector system needs to be connected and initialized:

```
auto s = createSystem("/etc/opt/xsp/system.yml");
s->connect();
s->initialize();
```

`connect()` establishes the control and data network links for all configured detectors and receivers, and the system changes into connected state.

`initialize()` performs the necessary hardware initialization, such as downloading calibration data, and the system changes into ready state. In addition, a READY user event is dispatched. Handling of user events is described in more detail in Section 3.9.

These methods can also be called on each individual `Detector`, `Receiver` or `PostDecoder` object. Please note, that connecting and initializing a detector does not automatically connect and initialize the associated receivers.

Prior to program termination, the detector system can be disconnected with

```
s->disconnect();
```

The call disconnects all configured detectors and receivers. This method can also be called on each individual `Detector` or `Receiver`. object. Please note, that disconnecting a detector does not automatically disconnect the associated receivers.

The `disconnect()` method is implicitly called in the destructor of the `System` object, so that it does not necessarily need to be called from the user application.

3.2. Reset

The `System` object provides the following methods to reset the detector system:

```
s->reset();
```

The `reset()` method sends a reset command to the detectors. The effect depends on detector type and is described in the appendix for the specific detector type.

3.3. State

`Detector`, `Receiver` and `PostDecoder` objects maintain an internal state, which can be queried using one of the following methods (shown for a detector, but similarly available for a receiver and post decoder):

```
auto d = s->detector("lambda");
bool connected = d->isConnected();
bool ready = d->isReady();
bool busy = d->isBusy();
```

`isConnected()` returns, whether the network connection has been established. `isReady()` returns, whether an object is ready for acquisition, and `isBusy()` returns, whether an object is busy with an acquisition.

If called on the `System` object, the return value is the logical and of the states of all detectors, post decoders and receivers. Thus, a system returns true for `isReady()` only, if all detectors, post decoders and receivers are in the ready state.

3.4. Acquisition Parameters

All `Detector` objects support at least two methods: set the number of frames to acquire and set the shutter time:

```
d->setFrameCount(std::uint64_t count);
d->setShutterTime(double time_ms);
```

Both methods usually support a limited range of values for their argument, which is described in more detail in the related Detector User Manual. Depending on the argument values, the methods may throw a `RuntimeError` exception with status code `BAD_ARG_OUT_OF_RANGE`.

The hardware specific `Detector` objects usually provide much more acquisition parameters, which are described in more detail in the appendix.

3.5. Acquisition Commands

The `System` object provides the following methods to control acquisition:

```
s->startAcquisition();
s->stopAcquisition();
```

The `startAcquisition()` method starts an acquisition on all detectors, and `stopAcquisition()` stops a running acquisition on all detectors. If the `startAcquisition()` method is called, while the detector system is in the busy state, a `RuntimeError` exception with status code `BAD_DEVICE_BUSY` is thrown.

`startAcquisition()` can also be called on the `Detector` object. If the system consists of only one detector, then calling `startAcquisition()` on the `Detector` object is equivalent to calling `startAcquisition()` on the `System` object. The same applies to the `stopAcquisition()` method.

Before starting an acquisition, applications need to check, whether the following prerequisite is fulfilled:

- RAM buffer allocated:

  The RAM buffer is allocated during initialization. The allocation is performed in the background in chunks of 16MB, where the first chunk is allocated immediately, so that the call to `initialize()` returns after the first chunk has been allocated, but before the complete RAM buffer is allocated. Thus, applications need to call `ramAllocated()` on the `Receiver` object afterwards to determine, whether this is the case.

  Since one chunk is always allocated, acquisitions may already be started directly after initialization, but with reduced buffer size.

The receivers internally count the received frames and will stop the acquisition automatically, if the number of frames to acquire has been received. In this case, it is not necessary to explicitly call `stopAcquisition()`.

Starting an acquisition will change all objects into the busy state. In addition, a START user event is dispatched by detectors and receivers. Stopping an acquisition will change all objects back to the ready state. In addition, a STOP user event is dispatched by detectors and receivers. Handling of user events is described in more detail in Section 3.9.

3.6. Readout

The `Receiver` and `PostDecoder` objects provide the following methods to handle read out of acquired frames:

```
auto r = s->receiver("lambda/1");
r->frameWidth();
r->frameHeight();
r->frameDepth();
r->framesQueued();
auto f = r->frame(1500);
r->release(f);
```

The `frameWidth()`, `frameHeight()`, and `frameDepth()` methods return the dimension and bit depth of each acquired frame.

The acquired frames are queued inside the receiver, and the `framesQueued()` method returns the number of currently queued frames.

The `frame()` method returns a pointer to a `Frame` object. A timeout in milliseconds can be specified and the method returns a null pointer, if the timeout has elapsed and no frame was queued within that time.

Once data has been processed, e.g. copied into the user application, the `release()` method must be called to remove the frame from the queue and to free up the resources occupied by the acquired frame.

Once a frame pointer has been returned by `frame()`, the following methods are available to get access to frame number, status, data, size and other meta data:

```
auto f = r->frame();
f->nr();
f->subframe();
f->connector();
f->status();
f->data();
f->size();
f->seq();
```

The frame number returned by `nr()` always starts at 1 and increases in steps of 1. The library ensures that the sequence of frames is maintained, even if packets were lost on the data link between detector and slave host. The frame status indicates, whether some or all packets for a frame were missing.

Depending on the operation mode of the detector, each exposure may generate several so called sub frames. In that case `subframe()` returns the subframe number.

Some detectors support more than one connected sensors, and `connector()` returns the connector number, on which a frame has been received. Usually, only one sensor is connected to the receiver, so that in most cases a 1 is returned.

`status()` returns the frame status, which can be one of `FRAME_OK`, `FRAME_INCOMPLETE`, `FRAME_MISSING`, and `FRAME_COMPRESSION_FAILED`.

The pointer returned by `data()` points to the memory region containing the frame data. The number of bytes within this memory region is returned by the `size()` method. Applications may need to cast the pointer to a different integer type, in case of frame data is stored as multi-byte values.

3.7. Live View

The `Detector` object provides the following methods to handle read out of live frames:

```
d->liveFrameWidth();
d->liveFrameHeight();
d->liveFrameDepth();
d->liveFramesQueued();
d->liveFrameSelect(1, 1);
auto f = d->liveFrame(1500);
d->release(f);
```

The `liveFrameWidth()`, `liveFrameHeight()`, and `liveFrameDepth()` methods return the dimension and bit depth of the live frame, which has all individual module frames stitched together according to the configured module positions and base rotations.

The acquired frames are queued inside the detector, and the `framesQueued()` method returns the number of currently queued live frames.

The `liveFrameSelect()` method filters the live frames based on the specified subframe and connector number. From the stream of acquired frames, only those frames matching the selected values are added to the live view update. A call of this method only changes the live frame update for the next acquisition run. A live view update from a running acquisition is not affected.

The `liveFrame()` methods returns a pointer to a `Frame` object. A timeout in milliseconds can be specified and the method returns a null pointer, if the timeout has elapsed and no live frame was queued within that time.

Once data has been processed, e.g. displayed in the user application, the `release()` method must be called to remove the live frame from the queue and to free up the resources occupied by it.

3.8. Logging

Users can install a log handler to get additional information during the execution. The code to add a log handler is shown below:

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

The handler is set with the `setLogHandler()` function. It requires a `std::function` object, which can for example be defined using a lambda expression. The

handler is called with 2 arguments: a log level, and a message. The above example handler prints warnings and errors to standard error, and all others to standard output.

The log handler must be removed, before it goes out of scope:

```
clearLogHandler();
```

3.9. User Events

The library dispatches user events, e.g. when a detector has been started or stopped. Applications can define functions to handle such events. The handlers are set on a `Detector`, `PostDecoder` or `Receiver` object. In the following example, the handler is set on the `Detector` object using the `setEventHandler()` method:

```
auto s = createSystem("/etc/opt/xsp/system.yml");
auto d = s->detector("lambda");
d->setEventHandler([&](auto t, const void* d) {
    switch (t) {
        case EventType::READY:
            // handle detector ready event
            break;
        case EventType::START:
            // handle detector start event
            break;
        case EventType::STOP:
            // handle detector start event
            break;
    }
});
```

The `setEventHandler()` method requires a `std::function` object as an argument, which can for example be defined using a lambda expression. The handler is called with 2 arguments: an event type, and a pointer to optional event data.

The following event types are defined:

- READY:

  The READY event is dispatched after all modules have been initialized and the detector is ready for acquisition. It is dispatched both by `Detector` and `Receiver` objects.

  This event does not carry additional data. The data pointer is set to `nullptr`.

- START:

  The START event is dispatched, if an acquisition is started with a call to `startAcquisition()`. It is dispatched by `Detector`, `PostDecoder` and `Receiver` objects.

  The event carries the number of frames to acquire. Handlers need to type cast the pointer passed in as the second argument to a pointer to `std::uint64_t`:

  ```
  void handler(EventType t, const void *d) {
      if (t == EventType::START) {
          auto n_frames = *static_cast<const std::uint64_t*>(d);
          // handle start event
      }
  }
  ```

- STOP:

The STOP event is dispatched, if an acquisition is stopped either with a call to `stopAcquisition()`, or if all frames have been received and the receiver automatically stops an acquisition. It is dispatched both by `Detector`, `PostDecoder` and `Receiver` objects.

This event does not carry additional data. The data pointer is set to `nullptr`.

4. Configuration

The configuration of a detector system is stored in a file, which is read upon creation of the `System` object. It contains overall settings for the system and specific settings for each detector and receiver. For multi-module systems, it also contains synchronization settings.

By default, the configuration file is stored according to Linux Filesystem Hierarchy Standard (FHS) in the directory `/etc/opt/xsp`.[1] The default name of the system configuration file is `system.yml`.

The configuration file is written in YAML.[4] The following example shows the system configuration file for a detector system with one single module Lambda detector:

```
# system.yml
system:
  id: BL18-2

detectors:
  - id: lambda
    type: Lambda
    # following settings are Lambda detector specific
    operation-mode:
      bit-depth: 12
    modules:
      - directory: Module_2017-007_Si
        control:
          ip: 169.254.1.2

receivers:
  - ref: lambda/1
    type: Lambda
    links:
      - ip: 192.168.11.1
        mac: a0:b3:fd:01:ff:00
      - ip: 192.168.12.1
        mac: a0:b3:fd:01:ff:01
```

The `system:` section defines the overall ID for this detector system, and optionally the IP address on the control network and host synchronization parameters.

The `detectors:` section defines the configuration of the detectors, which are connected to the system. Each detector is assigned a unique ID and a type. It also defines the IP address for the control interface.

The `receivers:` section defines the configuration of the data receivers. For each receiver a detector reference and the data connection parameters are configured. The detector reference is a string composed of the detector ID, a slash, and the module number. The module number can be omitted for single module detectors, in which case it is assumed to be 1.

4.1. System

The `system:` section defines the system identification (id).

```
system:
  id: BL18-2
```

The id is a single word, containing lower and upper case letters, numbers and the characters '-' and '_'. The ID can be omitted, in which case the default system id `SYSTEM` is assigned.

[1] This is usually a link to the directory `/opt/xsp/config`.

If multiple hosts are used to receive data, then additional `control:` and `sync:` mappings are needed to configure the synchronization. Complete examples for a master and slave host are shown below:

```
# master
system:
  id: BL18-2
  control:
    ip: 192.168.10.2
    timeout: 2000 # in milliseconds
  sync:
    slaves: [ 192.168.10.2, 192.168.10.3, 192.168.10.4 ]
    port: 4101   # sync port (default is 4101)
```

```
# slave
system:
  id: BL18-2
  control:
    ip: 192.168.10.2
    timeout: 2000 # in milliseconds
  sync:
    master: 192.168.10.2
    port: 4103   # sync port (default is 4103)
```

The `control:` mapping contains the IP address of the host on the control network and the timeout. The timeout is a value in milliseconds, after which sending or receiving messages over the control network is considered a failure. The timeout is optional, the default is 2000 ms.

The `sync:` mapping contains IP addresses of the master and slave hosts and the port to use for synchronization. `slaves:` must be specified on the master host, and `master:` must be specified on any slave host. The default synchronization port is 4101 for master hosts and 4103 for slave hosts. Another port can be specified using the `port:` mapping. If the non-default port is used on either master or slave side, then the port needs to be added to the IP address using *ip:port* notation. The following examples show the master and slave config files, where the master uses port 5001 and slave uses port 6001:

```
# master
system:
  […]
  control:
    ip: 192.168.10.2
  sync:
    slaves: [ 192.168.10.3:6001 ]
    port: 5001
```

```
# slave
system:
  […]
  control:
    ip: 192.168.10.3
  sync:
    master: 192.168.10.2:5001
    port: 6001
```

Important

Internally, the port range *port…port+1* is used. If multiple slave processed are executed on the same host, their synchronization ports need to differ at least by 2.

4.2. Detectors

Detector configuration is specified as a sequence using the '-' character. `detectors:` is optional, and is only present on a master host. The following example shows the configuration of a system with two Lambda type detectors:

```
detectors:
  - id: DET_01
    type: Lambda
    […]  # type specific config
  - id: DET_02
    type: Lambda
    […]  # type specific config
```

For each detector, an ID and type is specified. The ID must be a single word, containing lower and upper case letters, numbers and the characters '-' and '_'. The ID is used to reference a detector within a receiver configuration. The type specifies the detector type. Currently, only `Lambda` is supported.

Hardware specific configuration parameters are described in more detail in the appendix.

Live View

A detector provides a live view of acquired images at a default period of 1000 ms. The rate can be changed using the optional `live-update:` mapping:

```
detectors:
  - […]
    live-update: 1000   # ms
```

User Data

The detector configuration may contain additional user data that need to be available to applications, such as the detector serial number or sensor properties:

```
detectors:
  - […]
    user-data:
      serial_number: LACdTe750K2018-001
      sensor_material: Si
      sensor_thickness: 500   # micrometer
```

The optional `user-data:` mapping contains user defined key value pairs. The key is used to identify a pair, and the value is some free text. Numerical values are treated as text. A value can be followed with a comment, which is not part of the value.

One use case is the addition of particular information to image data files. For example, the NeXus file format contains data fields to store sensor material and sensor thickness. These fields are named *sensor_thickness* and *sensor_material*, and values can be retrieved by calling `userData()` API method of either the detector or receiver.

4.3. Receivers

Data receiver configuration is specified as a sequence using the '-' character. `receivers:` is optional, and is only present, if there is a data connection to the detector on that host. The following example shows the configuration for two Lambda type receivers:

```
receivers:
  - ref: lambda/1
```

```
    type: Lambda
    links:
      - ip: 192.168.11.1
        mac: a0:b3:fd:01:ff:00
      - ip: 192.168.12.1
        mac: a0:b3:fd:01:ff:01
  - ref: lambda/2
    type: Lambda
    links:
      - ip: 192.168.13.1
        mac: a0:b3:fd:07:a0:f0
      - ip: 192.168.14.1
        mac: a0:b3:fd:07:a0:f1
```

Detector Reference

Each receiver requires a reference to the connected detector module. It is configured with `ref:` and the reference is specified using the detector ID and a module number, separated with a '/'-character.[2] If a detector has only one module, then the module number can be omitted, and is assumed to be 1.

Data Links

The associated physical data links are configured as a sequence, since there can be more than one link between a detector module and host. Each link specifies IP address, MAC, and an optional port on the destination host. If port is not specified, then the default port 4601 is used. Also, an optional receive timeout can be specified. The default timeout is 2000 ms.

```
receivers:
  - […]
    links:
      - ip: 192.168.11.1
        mac: a0:b3:fd:01:ff:00
        port: 4601
        timeout: 2000
      - ip: 192.168.12.1
        mac: a0:b3:fd:01:ff:01
        port: 4601
        timeout: 2000
```

NUMA and CPU Affinity

On multiprocessor systems with NUMA, a receiver can be pinned to a specific NUMA node.[3] If pinned to a NUMA node, then the RAM buffer for storing the raw detector data is allocated in the memory local to that NUMA node. This is the preferable configuration, since access time to the NUMA local memory is smaller then access time to memory local to other NUMA nodes. It also uses only cores from that node for all decoding threads. The following example shows a receiver pinned to NUMA node 1:

```
receivers:
  - […]
    numa: 1
    links:
      - […]
      - […]
```

Each physical data link is handled by a dedicated read thread, which can be further pinned to a specific CPU core. This core should not be the one, which is configured to receive the interrupts from the network card, in order to avoid packet loss due to interrupts not handled fast enough. On NUMA hosts, the core should be on the node, as configured with `numa:`.

[2] Module numbers always start at 1.
[3] The available NUMA nodes can be listed using `numactl --hardware`.

Cores are numbered starting with 0. However, core #0 should be avoided, since it usually runs timer tasks.[4] The following example shows the `cpu:` mapping to pin read threads to specific cores. If the CPUs support hyperthreading, then two cores sharing the same L1 cache should be used.[5]

```
receivers:
  - […]
    numa: 1
    links:
      - […]
        cpu: 3
      - […]
        cpu: 15
```

Decoding

The following decoding parameters can be configured: the number of decoding threads and the amount of RAM to use for the intermediate storage of data packets. The amount of RAM is specified either as a relative percentage of total physical RAM, or an absolute number with GB (for gigabytes), MB (for megabytes), or kB (for kilobytes). A number without unit is counted as bytes.

```
receivers:
  - ref: lambda/1
    type: Lambda
    decoding:
      threads: 4
      ram-use: 50%  # or e.g. 32GB, 16000MB, 8000000kB
    […]
```

All parameters are optional.The default number of decoding threads is 4, and the default amount of RAM to use is 25%.

Note

The physical RAM, which is used for the intermediate storage of UDP packets is internally limited to a maximum of 80% of available system RAM.

If the receiver is pinned to a specific NUMA node, then memory is allocated local to that NUMA mode, and the decoding threads are pinned to the CPUs from that node only.

A timeout can be set for the frame aggregation, which is the process of merging all UDP packets into a single frame. The timeout value is specified in milliseconds. The frame aggregation timeout defines a period, after which a frame is considered incomplete, if not all packets for this frame arrived within this period.[6] The default value is 500ms.

```
receivers:
  - ref: lambda/1
    type: Lambda
    decoding:
      […]
      frame-timeout: 500
    […]
```

The decoding pipeline contains several queues to store intermediate frame data. The size can be adjusted using the `queue-size:` mapping, the value is in number of frames. The default length is 50 frames.

[4] This is a remark from the Intel X710 Tuning Guide [5].
[5] Processing units sharing the same L1 cache have the same physical id and core id, as reported by `/proc/cpuinfo`. The tool `lstopo` provides a graphical overview of how processing units are numbered in a multicore processor.
[6] Time restarts counting on reception of a packet for a specific frame.

```
receivers:
  - ref: lambda/1
    type: Lambda
    decoding:
      […]
      frame-queue: 100  # default: 50
    […]
```

Compression

Receivers can compress data on the fly. If no compression is specified, or the compressor is set to `none`, then data is not compressed. The library provides the zlib and blosc compressor.[6][7] The zlib compressor uses the deflate algorithm, whereas the blosc compressor can be configured to use different algorithms by adding the algorithm name after a slash, such as `blosc/blosclz`.[7] The compression level can be configured to a value between 0 and 9: 0 means to not compress and just copy, 1 optimizes for speed and 9 optimizes for size. The default level is 2.

`threads:` configures the number of compression threads to use. The blosc compressor may run its algorithm using even more threads, which are called internal threads, and those can be configured using `internal-threads:`.

The blosc compressor allows to shuffle data before compression. Shuffling can be on byte or bit level. In automatic mode, shuffling depends on the pixel width: if width is 8 bits or less, then data is not shuffled, if width is 16 bits or more then bytes are shuffled. Shuffling can be switched off by setting the value to `none`. The default is to not shuffle the data before compression.

```
receivers:
  - ref: lambda/3
    type: Lambda
    compression:
      compressor: blosc/blosclz   # alternatively: zlib
      level: 2
      threads: 4
      internal-threads: 6
      shuffle: none   # one of none, auto, byte, bit
    […]
```

UDP Buffer

For each receiver, an UDP receive buffer size may be specified.[8] The size can be given as a number with MB (for megabytes), or kB (for kilobytes). A number without unit is counted as bytes. The default UDP receive buffer size is 64MB:

```
receivers:
  - ref: lambda/3
    udp-buffer: 67108864     # 64MB, 65536kB
    […]
```

4.4. Post Decoders

An optional `postdecoders:` section defines post decoders to further process decoded frames.

```
postdecoders:
  - ref: lambda
    input:
      frame-timeout: 100
```

[7] The list of algorithms depend on the blosc installation,but is usually: blosclz,lz4,lz4hc,snappy,zlib,and zstd.
[8] The OS defines a maximum receive buffer size. Usually, this is set to 128MB, but it can be increased by the superuser with the command ‚sysctl -w net.core.rmem_max=<n bytes>‘.

```
    threads: 3
    compression:
      compressor: blosc/zlib
      level: 2
      shuffle: auto
      internal-threads: 4
    output:
      frame-queue: 100
# or
  - ref: lambda/*
    input:
      frame-timeout: 100
    frame-summing: 100
    output:
      frame-queue: 100
# or
  - ref: lambda/2
    input:
      frame-timeout: 3000
      processing-timeout: 2000
    threads: 1
    frame-summing: 500
```

Receiver Reference

Each post decoder needs a receiver reference, which is configured using the `ref:` key. The reference can be a single receiver in the form `detector_id/module_nr` or `detector_id`. In the first form, the post decoder is attached to a single receiver, in the second form, the post decoder is attached to all receivers of a detector.

Important

The second form only supports setups, where all detector modules are connected to a single server.

The first form also allows to use a wildcard instead of the module number, such as `detector_id/*` to define multiple identical post decoders that are each attached to the corresponding receivers.

Frame Processing

Processing of decoded frames includes:

- summing of consecutive frames into a single frame
- stitching frames from multiple modules into a single frame using the position information from each module
- compressing frames

The number of threads used to process frames can be set with the `threads:` key. The `frame-summing:` key defines the number of input frames to sum into a single output frame. A value of 1 disables frame summing. The default value is 1.

Important

Frame summing is not supported for dual counter mode and discrete sensor layout.

Note

If frame summing is enabled, i.e. the number of frames is larger than 1, then the bit depth changes to 32 bit regardless of the detector settings.

Stitching is automatically enabled for post decoders using the second form for their reference.

Post decoders can compress data on the fly. If no compression is specified, or the compressor is set to `none`, then data is not compressed. The library provides the zlib and blosc compressor.[6][7] The zlib compressor uses the deflate algorithm, whereas the blosc compressor can be configured to use different algorithms by adding the algorithm name after a slash, such as `blosc/blosclz`.[9] The compression level can be configured to a value between 0 and 9: 0 means to not compress and just copy, 1 optimizes for speed and 9 optimizes for size. The default level is 2.

The blosc compressor allows to shuffle data before compression. Shuffling can be on byte or bit level. In automatic mode, shuffling depends on the pixel width: if width is 8 bits or less, then data is not shuffled, if width is 16 bits or more then bytes are shuffled. Shuffling can be switched off by setting the value to `none`. The default is to not shuffle the data before compression.

Parameters

Several timeouts can be set inside in the `input:` section. A `frame-timeout:` specifies the timeout to wait for frames from the associated receiver. The default value is 1500ms. `processing-timeout:` is a timeout used internally while processing input frames from multiple modules, when stitching is enabled. The default value is 3000ms. The processing timeout shall be larger than the frame timeout. The output frame queue size can be set in the `output:` section using the `frame-queue:` key. The default frame queue size is 50.

[9] The list of algorithms depend on the blosc installation,but is usually: blosclz,lz4,lz4hc,snappy,zlib,and zstd.

A. Lambda

This chapter describes the configuration and programming interface, which is specific to Lambda type detectors. The first section defines some concepts and terms.

A.1. Definitions

A.1.1. Sensor Modules

Sensor modules exist in 2 variants:

- compound sensors
- discrete sensors

Compound sensor modules combine 1 to 12 read out chips with the sensor tiles in a fixed arrangement. Compound modules are directly attached to readout board. Examples for detectors with compound sensors are Lambda 750K, 250K and 60K.

Discrete sensor modules combine a single sensor tile with a single readout chip. They are connected to the readout board with wires. Example for a detector with discrete sensors is the Lambda flex.[1]

A.1.1.1. Compound Sensors

Compound sensor modules have two different sizes:

- one to carry 6 or 12 chips, and
- one to carry 1 or 4 chips.

The individual chips are numbered as shown in Figure A.1 and Figure A.2, when looking at the sensor. The shaded chips are the ones that are used, if not all chips are mounted. Custom 1-chip modules may have a readout chip connected to a different position.

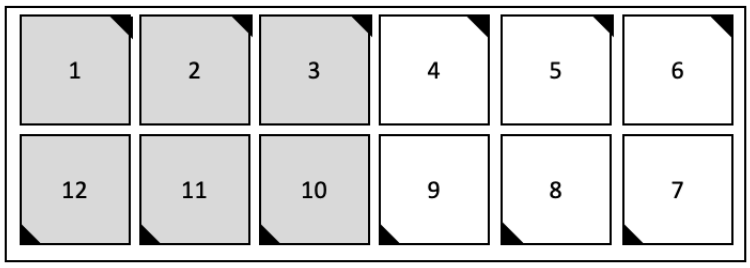

Figure A.1. Chip numbering on 6 and 12 chip module

[1] Lambda flex is the new name for Hydra.

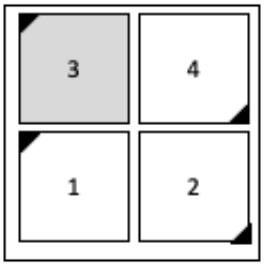

Figure A.2. Chip numbering on 1 and 4 chip module

Between the chips on a module, there is a gap of several pixels in both horizontal and vertical direction. On recent modules, this gap is 4 pixels in both horizontal and vertical direction. Some older modules exist, that have different number of gap pixels in horizontal and vertical direction.

Gap pixels form with their adjacent border pixel of the chip a so called extra large pixel, because the sensitive area is larger than that of a regular pixel. The charge collected in this area is equally distributed to the border pixels of the two adjacent chips.

The sensitive area can be reduced by adding a guard above the gap, which removes the charge in this area. In that case, the border pixels count as regular pixels.

Recent 6-chip modules have four unconnected pixel columns on one side in order to combine two of them to a 12-chip module.[2] These columns are on the right side of chips 3 and 10 in Figure A.1. Some old 6-chip modules exist, where this is not yet the case.

A.1.1.2. Discrete Sensors

Discrete sensor modules combine a single readout chip with a single sensor, as shown in Figure A.3, when looking at the sensor with the wirebonds at the bottom. Up to 4 discrete sensor modules can be connected to a 250K readout board. The actual chip number is defined by the connector number *c*.

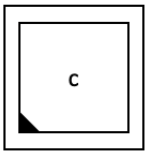

Figure A.3. Chip numbering on discrete sensor module

A.1.2. Frames

Sensor readout is composed into frames. For compound sensors, data from all chips are combined into one compound frame. For discrete sensors, chip data is put into separate discrete frames with one frame per chip.

In dual-threshold mode, the number of frames is doubled, where the first compound frame or the first set of discrete frames contain the counts using the first threshold, and the second compound frame or set of discrete frames contain the counts using the second threshold.

[2] Si modules are built with a single sensor tile, connected to 12 readout chips. GaAs and CdTe sensors are built with two sensor tiles, each connected to 6 readout chips.

A.1.2.1. Compound Frame

The frame is a two-dimensional array of unsigned integers, stored in row-major order. Row index increases along the x-axis, and column index increases along the y-axis. The first stored array value is the pixel value at (x,y)=(0,0), which is the top left corner of the frame. Depending on the configured detector bit depth, pixel values are stored as specified in the following table:

Table A.1. Frame data pixel value type

Detector Bit Depth	C++ data type
1	uint8_t
6	uint8_t
12	uint16_t
24	uint32_t

X-ray beam is in the direction of positive z-axis. The frame is stored as viewed from the interaction point towards the detector in the direction of the beam.

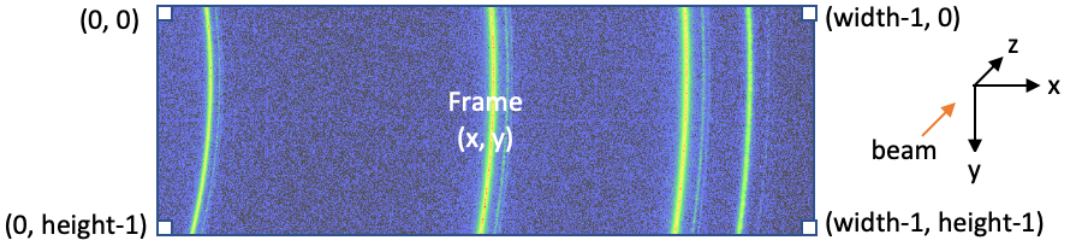

Figure A.4. Frame coordinate system

Gap pixels are part of the frame data. If they are not guarded, then their count is interpolated using the value of the adjacent extra large pixel, as shown in Figure A.4. At the edges, the count is divided by 3 and is reassigned to the extra large pixel and underlying gap pixels. At the corners, the count is divided by 9.

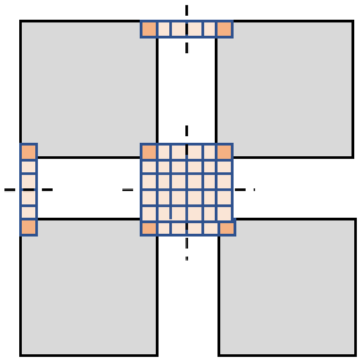

Figure A.5. Intra module gap pixels

12-chip modules can be built using one full or two half sensor tiles. In case of two half tiles, there is an additional gap between the tiles equal to 2 pixels. Together with the unconnected pixel columns on the half tiles, this results in a number of 10 black pixels without a count in the middle of the frame. On 6-chip modules, the unconnected pixels at the border are not part of the frame data.

The frame size for different sensor modules is summarized in the following table.

Table A.2. Frame size of sensor modules

Module	Size (width x height)
12-chip Si module (1 full sensor)	1556 x 516
12-chip GaAs/CdTe module (double hexa) with two times 4 unconnected columns plus 2 gap pixels between tiles	1554 x 516
6-chip module (single hexa) with 4 unconnected columns on right border	772 x 516
4-chip module	516 x 516
1-chip module	256 x 256

A.2. Configuration

A Lambda detector configuration extends the basic detector configuration, which is described in Section 4.2. The following example shows a typical configuration for a single module detector, such as the 750K Si:

```
detectors:
  - id: lambda
    type: Lambda
    control:
      ip: 169.254.1.2
      port: 4321
    modules:
      - directory: Module_2017-007_Si
    operation-mode:
      bit-depth: 12
```

A.2.1. Control Link

For each detector, the control link IP address, port and timeout in milliseconds are configured as follows:

```
detectors:
  - […]
    control:
      ip: 169.254.1.2
      port: 4321
      timeout: 2000
```

All values are optional. The default IP address of the detector's control interface is 169.254.1.2. The default port is 4321. The default timeout is 2000 ms.

A.2.2. Modules

The `modules:` section contains module specific configuration. Module numbers are assigned automatically in the sequence as they appear in the configuration file. The first module gets the number 1, the second the number 2, and so on.

Module calibration data is read from a directory, which can either be specified in the configuration file, or which is derived from the detector ID plus '_'-character plus the module number. Directories can be relative to the location of the system configuration file, or absolute.[3] The following excerpt shows an example of a two module detector with explicit module directory paths:

```
detectors:
  - […]
    modules:
      - directory: Module_2019-010_Si
      - directory: Module_2019-011_Si
```

[3] A common use case for absolute paths is a central storage for detector configurations.

For multi module detectors, each module requires the specification of a position and rotation relative to the detector origin enclosed in curly brackets:

```
detectors:
  - […]
    modules:
      - […]
        position: { x: 0.0, y: 0.0, z: 0.0 }
        rotation: { alpha: 0.0, beta: 0.0, gamma: 0.0 }
```

Origin is top left corner, when looking in the direction of the beam. The x coordinate is along the top edge of the housing with positive values to the right, the y coordinate is along the side edge of the housing with positive values to the bottom, and the z coordinate is in the direction of the beam.[4] The first rotation angle `alpha:` specifies a rotation around the z-axis between the x-axis and the rotated x'-axis, with positive values for a rotation towards the y-axis.

Note

The rotation specified with `alpha:` is a corrective rotation with respect to the base orientation of the sensor module. A sensor module can be operated in horizontal (rotation by 0 or 180 degrees) or vertical orientation (rotation by 90 or 270 degrees). The base orientation of a sensor module is specified in the `sensor:` mapping as described in Section A.2.7.

Newer modules since firmware version 2.0.0 have a programmable high voltage, which needs to be defined in the configuration per module using the `hv:` mapping. The value is a positive floating point number.[5] The default value is 0.0. The maximal absolute value can be adjusted with an optional configuration mapping `max-hv:`. The default maximum values are 200.0V for Si and 300.0V for HiZ. If the `hv:` voltage is configured larger than the maximum voltage, it is silently limited to the maximum when programmed into the detector. Depending on the firmware version, programming the voltage may take some time.[6] In order to detect a failure while setting the voltage, a variable timeout can be set with `hv-timeout:`. The default timeout is 12s. Newer firmware has been improved to ramp the voltage in the background, so that the timeout is not needed any more. Please look at the `setVoltage()` method in the API Reference Manual for more information.

```
detectors:
  - […]
    modules:
      - […]
        hv: 200.0
        max-hv: 200.0  # default is 200.0 for Si
        hv-timeout: 12 # default is 12s
```

An optional `timing:` mapping allows to adjust readout timings per chip:

```
detectors:
  - […]
    modules:
      - […]
        timing:
          # for generation 1 boards, delays are per chip
          - chip: 3
            delays: [ 1, 2, 1, 3, 1, 2, 2, 1, 4 ]
          # for generation 2 boards, delays are per module half, no chip: needed
          - delays: [ 1, 2 ]
```

[4] See also definition of frame coordinate system as shown in Figure A.4

[5] Si modules require a positive high voltage, and HiZ modules require a negative high voltage. The sign however is defined by the voltage regulator on the readout board and cannot be changed via configuration. Thus configuration only specifies absolute values.

[6] The amount of time used to ramp the voltage to the configured value depends on the Z-diode, used on the readout board. Typical values are 7s for going to 200.0V, and 10s for going to 300.0V

```
    […]
```

For generation 1 boards, nine delays have to be specified: one for each of the 8 readout lines, and one for the clock line. Values may be between 0 and 63. Not all detectors allow adjustment of the timing, and this setting is silently ignored if not supported.

For generation 2 boards, only two delays have to be specified: one for the left half and one for the right half. Since the delays are per module, the chip number does not need to be specified. It is internally assumed to be 0 in that case.

Important

The firmware of generation 2 boards now uses information stored in the file system on the detector to adjust the I/O timing. Having these values in the configuration file is no longer required and thus deprecated.

Each module of a multi-module detector may have its own control interface, in which case the IP addresses are specified on module level, as shown in the following example:

```
detectors:
  - id: lambda
    type: Lambda
    modules:
      - control:
          ip: 169.254.1.2
          port: 4321
        […]
      - control:
          ip: 169.254.1.10
          port: 4321
        […]
      - control:
          ip: 169.254.1.18
          port: 4321
        […]
```

Important

The control IP address must be specified either on detector level, or for all modules on module level. The IP addresses on module level take precedence over any IP address configured on detector level.

For each module, a set of flags can be defined to alter their runtime behaviour. The 'ignore-errors' flag instructs the library to ignore errors returned by the firmware. In such a case, still a warning is logged, but no exception is thrown.

Flags can be defined either as a sequence using the '-' character, or as a list of strings in square brackets:

```
detectors:
  - id: lambda
    type: Lambda
    modules:
      - flags:
          - ignore-errors
        flags: [ "ignore-errors" ]   # alternative syntax
        […]
```

A.2.3. Master

Multi-module detectors may have an additional master board to synchronize acquisition across all modules. When acquisition is started or stopped, the final

command is sent to this board. If no separate master board is defined, then the first defined module is assumed to operate as master and all other modules are assumed to operate as slaves. The start command is sent first to all slave modules and then to the master module.

The only configurable parameter for the master board is the IP address and port of the control link:

```
detectors:
  - id: lambda
    type: Lambda
    master:
      control:
        ip: 169.254.1.2
        port: 4321
    modules:
      - […]
        control:
          ip: 169.254.1.10
      - […]
        control:
          ip: 169.254.1.18
      […]
```

Important

When defining a master board, no control link may be defined on detector level. Thus, all modules are required to have a control link configuration.

A.2.4. Operation Mode

The detector configuration contains initial operation mode to use upon start up:

```
detectors:
  - […]
    operation-mode:
      bit-depth: 12             # 1, 6, 12, 24
      charge-summing: off       # on, off
      counter-mode: single      # single, dual
      pitch: 55                 # 55, 110
      gain-mode: superhigh      # superhigh, high, low, superlow
      polarity: holes           # electrons, holes
```

The key `bit-depth:` defines the number of bits for each frame pixel.

The key `counter-mode:` specifies the number of counters to readout for one frame. If set to dual, both counters are read out for one exposure. The threshold for both counters is set to different values to give different counts, so that this mode of operation is called *dual threshold* mode. The resulting frame is calculated by subtracting both count values.

All operation mode keys are optional. The default values are equal to the values shown in the above example.

A.2.5. Calibration Data

Calibration data is read from files in the specified module directory. There is one file named `calibration.yml` for the DAC settings, and a set of *.bin files for the pixel threshold settings.

Calibration data may differ depending on whether charge summing is on (charge summing mode) or off (single pixel mode). Thus, two sets of calibration data may be defined: a single pixel mode (SPM) set and a charge summing mode (CSM) set. Calibration data is loaded as follows:

1. if <*module directory*>/spm/calibration.yml exists, then calibration data set for SPM is loaded from spm subdirectory
2. if <*module directory*>/spm/calibration.yml does not exist, but <*module directory*>/calibration.yml exists, then calibration data set for SPM is loaded from the module directory
3. if <*module directory*>/csm/calibration.yml exists, then calibration data set for CSM is loaded from the csm subdirectory

If only one of the calibration data sets is defined, then it will be used in both SPM and CSM.

Note

Pixel threshold settings are only loaded, if there is a file calibration.yml in the same directory.

DAC settings are read from a file calibration.yml. The file contains default settings and then per chip settings that differ from the defaults:

```
defaults:
  dacs:
    I_Preamp: 100
    I_Ikrum: 5
    I_Shaper: 125
    I_Disc: 150
    I_Disc_LS: 25
    V_Rpz: 255
    I_TP_BufferIn: 5
    I_TP_BufferOut: 128
    V_Tp_ref: 120
    V_Tp_refA: 50
    V_Tp_refB: 255
chips:
  - nr: 1
    dacs:
      I_DAC_DiscL: 82
      I_DAC_DiscH: 92
      V_Fbk: 175
      V_Cas: 181
      V_Gnd: 139
    thresholds:
      base: [ 10.0, 9.98 ]
      slope: [ 5.8333, 6.7112 ]
  - nr: 2
    […]
```

Chip numbers follow the scheme as shown in Figure A.1 and Figure A.2. Per chip, the dacs: settings are specified using the DAC names as specified in [8]. The thresholds: setting specifies a maximum of eight values each for base and slope, which are used to convert energies given in keV into digital threshold values according to the following formula. Currently only the first 2 thresholds are equalized, so that it is sufficient to specify only 2 values for base and slope.

$$threshold_n = [\mathrm{energy} \cdot slope_n + base_n], with n = [0,7]$$

The resulting digital value is then cropped to a maximum value of 511.

Note

The calibration file may contain only the defaults: settings, in which case the chips are autodetected by trying to read each chip ID.

Pixel threshold settings are read from binary files:

- for ConfigDiscL: `Chip#_THAsetting.bin`
- for ConfigDiscH: `Chip#_THBsetting.bin`
- for MaskBit: `Chip#_mask.bin`

where `#` is translated into a chip number `1` to `12`. Each value is stored as 16-bit integer in big endian order and row-major order. However, within each row, the pixel settings are stored from column 255 to 0 (thus in reverse order). File length is 256*256*2=131072 bytes.

A.2.6. Decoding Settings

The overall decoding settings are defined in the `decoding:` mapping:

```
detectors:
  - […]
    decoding:
      flatfield-correction: on      # on, off
      countrate-correction: on      # on, off
      saturation-flag: on           # on, off
      saturation-threshold: 200000  # counts/s/pixel
```

They are explained in more detail in the following sections.

A.2.6.1. Flatfield Correction

The key `flatfield-correction:` defines whether flatfield correction shall be applied or not. If set to `on`, the counts are corrected by multiplication with flatfield correction values. The default value is `off`.

The flatfield correction values are searched in a list of directories, which are defined per module using `flatfield-path:`. The syntax is shown below:

```
detectors:
  - […]
    modules:
      - […]
        decoding:
          flatfield-path: [ /path1/to/flatfields, /path2/to/flatfields, default ]
          # or
          flatfield-path:
            - /path1/to/flatfields
            - /path2/to/flatfields
            - default
```

If no path is defined, then the default search path is a directory named `flatfields` inside the module directory. If paths are specified in the configuration file, then the default search path is no longer used. Thus, if the modules flatfield directory should still be searched, it has to be explicitly added using the keyword `default`.

Inside each directory of the search path, there can be subdirectories `spm` and `csm` to keep separate sets of flatfield correction values for single pixel mode and charge summing mode, or the flatfield correction values can be stored directly into that directory, in which case they are used in both modes. Correction values from the `spm` and `csm` directories take precedence over any values stored in the directory itself.

The actual data files are stored in subdirectories for each beam energy, at which the flatfields have been measured. Each data file contains the correction values for a specific threshold. An example structure is shown below:

```
/
```

```
+-- spm/
|    +--- flatfield.yml
|    +--- energy_20.0_keV/
|    |     +--- flatfield_en20.0_th8.0_keV.bin
|    |     +--- flatfield_en20.0_th10.0_keV.bin
|    |     +--- flatfield_en20.0_th12.0_keV.bin
|    |
|    +--- energy_30.0_keV/
|          +--- flatfield_en30.0_th13.0_keV.bin
|          +--- flatfield_en30.0_th15.0_keV.bin
|          +--- flatfield_en30.0_th17.0_keV.bin
|
+-- csm/
     +--- flatfield.yml
     + ...
```

The file `flatfield.yml` contains the file and directory names as well as additional meta data, such as the date and time of measurements and the author. An example is given below:

```
# flatfield.yml
author: X-Spectrum
timestamp: Tue, 30 Mar 2021 09:13:41 +0200
energies:
  - energy: 20.0  # keV
    margin: 1.5
    directory: energy_20.0_keV
    timestamp: Tue, 30 Mar 2021 09:13:41 +0200
    thresholds:
      - threshold: 8.0
        file: flatfield_en20.0_th8.0_keV.bin
      - threshold: 10.0
        file: flatfield_en20.0_th10.0_keV.bin
      - threshold: 12.0
        file: flatfield_en20.0_th12.0_keV.bin
        timestamp: Thu, 15 Apr 2021 15:47:27 +0200
  - energy: 30.0
    […]
```

Author is a free format text naming the person or company, which performed the flatfield measurement. It can only be set globally. The date and time is specified according to RFC 2822. It can be set globally, per energy or per threshold.

The energies and thresholds, at which the flatfields have been measured, are specified in keV. In addition, an energy margin in keV can be specified, which is used to decide how to calculate the resulting flatfield correction values. If not specified, the default margin is 1.5 keV. If the actual energy is within this margin around a measured energy, then that flatfield correction values are used unmodified. If the actual energy is outside this margin, then the resulting flatfield correction values are calculated by interpolating between the values from the next lower and upper measured energy. If the energy is within the margin of two energies, then the correction values from the lower energy is used.

If correction values need to be interpolated, then the actual threshold is also scaled to the two energies used for the interpolation. For example, if there are measurements for 20.0 and 30.0 keV, but the actual beam energy is at 25.0 keV, then a threshold of 12.5 keV is scaled to 10.0 keV to find the correction values for the lower energy and to 15.0 keV to find the correction values for the upper energy. The interpolation is then performed pixel by pixel based on the correction value measured at the lower energy. Author and timestamp is copied from the lower measurement.

If there are no correction values for the actual threshold, then the threshold is rounded half up to the nearest threshold, for which correction values exist, and these values are selected. Using the example from above, if at beam energy 20.0

keV the threshold is set to 11.0 keV, then the flatfield measured at threshold 12.0 keV is used.

The flatfield correction data file contains the correction coefficients for each pixel stored in row-major order. Each coefficient is a double-precision floating point number according to IEE754 (8 bytes long), stored in little-endian order. There is no specific requirement to use suffix `.bin`, since filenames are fully configured in the `flatfield.yml` file. However, the suffix used for data files generated by X-Spectrum is `.bin`.

A.2.6.2. Count Rate Correction

The key `countrate-correction:` defines whether counts shall be corrected or not. If set to `on`, the counts are corrected using a lookup table. The default value is `off`.

The values for the lookup table are read per module from a file `countrate.lut`, stored in the module directory. This file contains lookup values for each actual count normalized to a shutter time of 1s. The lookup values are stored as double-precision floating point number according to IEE754 (8 bytes long), stored in little-endian order, starting with the lookup for count 0, then 1, and so on. If an actual count is larger than the number of values in the lookup table, the count is set to the maximum lookup value.

Lookup tables may differ depending on whether charge summing is on (charge summing mode) or off (single pixel mode). Thus, two tables may be defined: a single pixel mode (SPM) table and a charge summing mode (CSM) table. Lookup tables are loaded as follows:

1. if *<module directory>*`/spm/countrate.lut` exists, then lookup table for SPM is loaded from `spm` subdirectory
2. if *<module directory>*`/spm/countrate.lut` does not exist, but *<module directory>*`/countrate.lut` exists, then lookup table for SPM is loaded from the module directory
3. if *<module directory>*`/csm/countrate.lut` exists, then lookup table for CSM is loaded from the `csm` subdirectory

If only one of the lookup tables is defined, then it will be used in both SPM and CSM.

A.2.6.3. Saturation Flag

The key `saturation-flag:` defines whether saturation should be flagged or not. If set to `on`, the MSB of each pixel is set, if the count for that pixel exceeds the saturation threshold. The default value is `off`.

The global saturation threshold can be set with `saturation-threshold:`. It is also possible to override the global saturation threshold setting per module, or to specify a file with thresholds per pixel:

```
detectors:
  - […]
    modules:
    - […]
      decoding:
        saturation-threshold: 200000  # counts/s/pixel
        saturation-file: saturation_thresholds.uint32
```

The saturation threshold file can be specified with a path relative to the module directory or with an absolute path. It contains values per pixel in row-major order, as 32-bit unsigned integer in big endian format.

The saturation thresholds from a file have precedence over per module and global settings.

A.2.7. Sensor

An optional `sensor:` mapping contains the configuration of the sensors, which can either be specified on detector level, as in the following example, or on a per module level:

```
detectors:
  - […]
    sensor:
      layout: full
    modules:
      - […]
```

`layout:` specifies the layout of the sensor tile, if it differs from the default. Possible values for `layout:` are shown in Table A.3.

Table A.3. Sensor Layouts

Layout	Description
default	default sensor layout: - full for 1-chip, 4-chip, and 12-chip modules - single for 6-chip modules
full	one sensor tile with 1, 4, or 12 readout chips
single	one sensor tile with 6 readout chips
double	two sensor tiles with 6 readout chips each
discrete	one sensor tile per readout chip

More details on the specific sensor layouts can be found in Section A.1.1.

Note

12-chip GaAs and CdTe modules are built with two single tiles and are therefore using a non-default sensor layout. These modules require a `sensor:` mapping, as shown in the following example:

```
detectors:
  - […]
    sensor:
      layout: double
    […]
```

An optional `base-rotation:` can be specified, if the base orientation of the sensor module is not horizontally, but instead rotated by either 90, 180, or 270 degrees. The resulting frames are then also rotated during the decoding to reflect the base orientation.

The rotation is clockwise around the z-axis, as defined by the coordinate system in Figure A.4. Possible values are 0, 90, 180, and 270. the default is 0:

```
detectors:
  - […]
    modules:
      - […]
        sensor:
          base-rotation: 90  # 0, 90, 180, or 270
    […]
```

Note

Any module rotation, as specified for `alpha:` in the modules `rotation:` mapping, is relative to the base rotation of the sensor as specified here.

The absolute rotation with respect to the horizon along the x-axis is the sum of both rotation angles.

A.2.8. Pixel Mask

The pixel mask is read from the file specified in the `pixel-mask:` mapping. The file name can be specified with a path relative to the module directory or with an absolute path. Data is stored as 32-bit integers in big endian order per pixel in row major order. In the example below, the pixel mask is read from a file `Module_2017-007_Si/pixelmask.uint32` relative to the system configuration file:

```
detectors:
  - […]
    modules:
      - directory: Module_2017-007_Si
        pixel-mask: pixelmask.uint32   # path relative to the module directory
```

The pixel mask contains 32 bit per pixel, with the meaning specified by the NeXus International Advisory Committee (NIAC). Bits 9 and 10 are reserved for internal use by X-Spectrum. The full list of bits is shown in the following table.

Bit	Description
0	gap (pixel with no sensor)
1	dead
2	under responding
3	over responding
4	noisy
5	undefined
6	pixel is part of a cluster of problematic pixels (bit set in addition to others)
7	undefined
8	user defined mask (e.g. around beamstop)
9	reserved by X-Spectrum
10	reserved by X-Spectrum
11-30	undefined
31	virtual pixel (corner pixel with interpolated value)

A.3. Decoding

Decoding of module data involves the steps, as depicted in Figure A.6

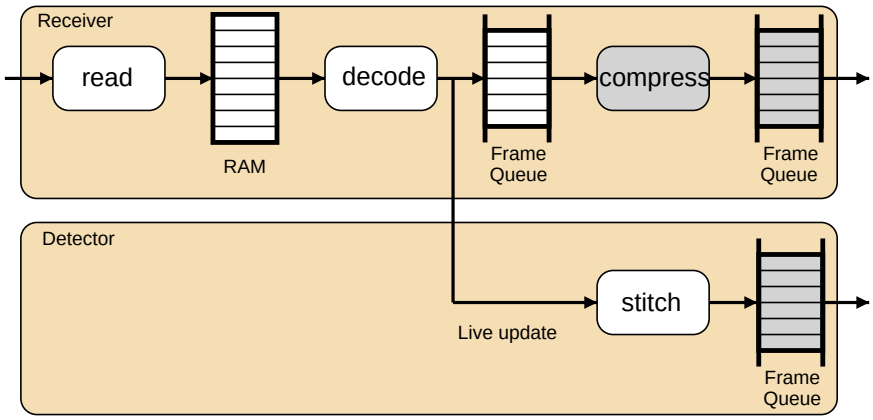

Figure A.6. Decoding steps

Incoming UDP packets are first stored into the large RAM buffer. Each packet contains a header with a packet and frame ID in order to sort incoming data. When all packets for a frame have been received, that frame is then passed to the decoding step.

Decoding works in one of two modes:

- normal mode
- test mode

In **normal mode**, the raw data from the detector are decoded into a frame, containing the counts per pixel. Depending on whether enabled or not, saturated pixels are marked, and counts are corrected using flatfield correction values. As a final step, the extra large pixels are interpolated, and the frames are stored into the frame queue. In regular intervals, one of the decoded frames is extracted for the live view. These live view updates from individual modules are sent to the detector object, where they are stitched into a full frame based on the configured position information.

In **test mode** the data is decoded chip wise and there is no further correction or interpolation applied to the counts. Live frames are not extracted. This mode is used only for calibration and test.

If configured, the decoded frames are also compressed.

After an acquisition has been started, the user application can read the decoded frames from the receiver object and the live frames from the detector object. Depending on whether compression is enabled, frames are read from either the receiver queue after the decode step or after the compress step.

A.3.1. Decoded Frames

Each decoded frame has the following associated meta data

- frame number
- subframe number
- connector number
- hardware sequence number

The frame number always starts at 1 for each acquisition. It is incremented by 1 for each exposure.

In dual counter mode, two subframes are decoded per exposure, where the first subframe has subframe number 1 and contains the counter values from the lower threshold, and the second subframe has subframe number 2 and contains the counter values from the higher threshold.

For discrete sensor layouts, as many chip frames are decoded per exposure, as there are chips connected to the readout board. All chip frames from one exposure have same frame number and the associated connector number.

The hardware sequence number is the number generated by the hardware for each frame sent via data link. It is not guaranteed that this number is reset to 1 for each acquisition. After power up of the detector, this number starts at 1 and is then incremented with each frame sent. In 24-bit mode, two 12-bit frames are sent, and both have different sequence numbers. The sequence number of the frame containing the lower 12 bits is stored as meta data into the decoded

24-bit frame. In dual counter mode, two 12-bit frames are sent, and both have different sequence numbers. For discrete sensor layouts, a single frame with data from all chips is sent, and the meta data for all chip frames contain the same hardware sequence number.

A.3.2. Saturation Flag

Pixels can saturate at a specific number of counts per second per pixel. In order to identify saturated pixels, they can optionally be flagged, if a specific count is above the defined saturation threshold. The threshold is specified in counts per second per pixel and will be scaled to the actual shutter time internally before the comparison.

The flagging takes place only for bit depths 24, 12, and 6. The flag is stored in the MSB of the pixel value. For depth 24, the flag is stored in bit 31, for depth 12, the flag is stored in bit 15, and for depth 6 the flag is stored in bit 7.

A.4. Programming Interface

The Lambda detector and receiver have an extended set of parameters to set. To get access to the extended API, the `Detector` and `Receiver` objects need to be casted to a pointer of type `xsp::lambda::Detector` and `xsp::lambda::Receiver` respectively:

```
#include <libxsp.h>
using namespace xsp;

int main()
{
    auto s = createSystem("/etc/opt/xsp/system.yml");
    auto d = std::dynamic_pointer_cast<lambda::Detector>(
        s->detector("lambda"));
    auto r = std::dynamic_pointer_cast<lambda::Receiver>(
        s->receiver("lambda/1"));
```

This section gives an overview of available methods, detailed description on each method can be found in the API reference manual.

A.4.1. General Information

The `xsp::lambda::Detector` object provides a couple of methods to return general information:

Number of Modules

The number of modules per detector is returned by

```
    auto n_modules = d->numberOfModules();
```

Firmware Version

The firmware version is returned per module by

```
    auto module_nr = 1;
    string fw_version = d->firmwareVersion(module_nr);
```

The return value is an informative string with firmware version, protocol versions, and feature bits.

Firmware is versioned using 3 numbers in the form "gen-type-rev". *gen* denotes the readout board generation, *type* denotes the firmware type, and *rev* denotes the revision of a specific firmware type.

The firmware type is a bit field with the following meaning:

Bit	Description
0	board type: 0=350K/750K, 1=60K/250K
1	software type: 0=bare metal, 1=Linux
2-5	reserved
6	master flag: 0=regular readout board (with module), 1=master board (without module)
7	slave flag: 0=regular readout board, 1=slave board (requires separate master, only for 750K generation 1 boards)

Chip IDs

The chip IDs is returned per module by

```
auto module_nr = 1;
vector<string> chip_ids = d->chipIds(module_nr);
```

The chip IDs are returned as a vector of 12 strings, for non-existing chips the ID is set to the empty string. The string contains a comma separated list of batch number, waver number and waver coordinates, the fab, and the raw integer value.

Features

Whether a certain feature is supported by the firmware can be tested with

```
auto module_nr = 1;
bool has_6bit_mode = d->hasFeature(module_nr, lambda::Feature::FEAT_1_6_BIT);
```

The arguments to the `hasFeature()` method are module number and an enum of type `xsp::lambda::Feature` which can have the values `FEAT_HV`, `FEAT_1_6_BIT`, `FEAT_MEDIPIX_DAC_IO`, and `FEAT_EXTENDED_GATING`.

Module Flags

For each module, certain flags can be set, either via configuration file as described in Section A.2.2, or with the method `enableModuleFlag()`. Flags can be cleared at runtime using `disableModuleFlag()`:

```
auto module_nr = 1;
d->enableModuleFlag(1, xsp::lambda::ModuleFlag::IGNORE_ERRORS);
```

The flag is of type `xsp::lambda::ModuleFlag` and has only one value `IGNORE_ERRORS`. If enabled, the library ignores errors from that module, so that an operation is not interrupted with an exception.

Operating Parameters

Users can read several operation parameters from a module, such as temperature, humidity, sensor current and high voltage (HV).

```
auto module_nr = 1;
auto temperatures = d->temperature(module_nr);
auto humidity = d->humidity(module_nr);
auto sensor_current = d->sensorCurrent(module_nr);
auto hv = d->voltage(module_nr);
d->setVoltage(module_nr, 200.0);
```

Temperature is returned as a vector of doubles, the number of entries depend on firmware version. For readout boards with firmware 2.x.x, three values are returned: the board temperature, the FPGA temperature, and the temperature

from the humidity sensor. Old boards with firmware version 0.0.0 do not support this command, and an empty vector is returned.

On newer firmware, the high voltage is ramped up as a background process, so that applications need to poll the detector to determine, whether high voltage has reached the programmed setpoint:

```
    auto module_nr = 1;
    while (!d->voltageSettled(module_nr)) sleep(1);
```

RAM Buffer

The capacity of the RAM buffer in number of frames can be determine with

```
    auto max_frames = r->maxFrames();
```

This is the number of frames that can be stored at the same time inside the buffer without creating an overflow.

A.4.2. Module State

In addition to the general state, as explained in Section 3.3, a module specific state can be retrieved:

```
    auto module_nr = 1;
    auto mod_1_connected = d->isModuleConnected(module_nr);
    auto mod_1_ready = d->isModuleReady(module_nr);
    auto mod_1_busy = d->isModuleBusy(module_nr);
```

A.4.3. Decoding Parameters

The decoding process can be adapted by enabling or disabling certain steps, which are applied to the decoded frame in this sequence:

- flag saturated pixels
- correct countrate
- interpolate extra large pixels
- apply flatfield correction
- compression

Initially, the steps are disabled except for interpolation, which is always enabled. Users may enable individual steps in the configuration file, as described in Section A.2.6:

```
detectors:
  - […]
    decoding:
      saturation-flag: on           # on, off
      countrate-correction: on      # on, off
      flatfield-correction: on      # on, off
```

Interpolation cannot be disabled via configuration file.

Compression is configured in the `receivers:` part of the configuration file as described in the section called “Compression”:

```
receivers:
  - ref: lambda/1
    type: Lambda
    compression:
```

```
    compressor: zlib
    level: 2
    threads: 4
[…]
```

Saturation Flag

When saturation flag is enabled, the MSB of each pixel is set, if the count is above the saturation threshold. The threshold is given in counts per pixel per second, which is then scaled to the shutter time.

Flagging can be enabled or disabled at runtime with the methods `enableSaturationFlag()` and `disableSaturationFlag()` on the detector object. The default is that saturation flag is disabled. Whether saturation flag is enabled can be checked with `saturationFlagEnabled()`, which is available on the detector and receiver object.

The saturation threshold can be specified globally or per pixel, either in the configuration file as explained in Section A.2.6.3, or at runtime with the methods `setSaturationThreshold()` and `setSaturationThresholdPerPixel()` on the detector object. The Lambda receiver object also provides methods to retrieve the saturation thresholds, and whether flagging is enabled or not:

```
d->setSaturationThreshold(200000);
d->enableSaturationFlag();

if (r->saturationFlagEnabled()) {
    auto sat_th = r->saturationThreshold();
    // process saturation flag ...
}
```

Countrate Correction

When countrate correction is enabled, the count values from the detector are replaced by values from a lookup table, which is stored in the module directory, as explained in Section A.2.6.2. The lookup table contains replacement counts for each detector count starting with 0 up to a maximal count. If the detector count is above the maximal value in the lookup table, then the replacement value of the maximum count is taken.

Countrate correction can be enabled or disabled either in the configuration file or at runtime with the methods `enableCountrateCorrection()` and `disableCountrateCorrection()` on the detector object. The default is that countrate correction is disabled. Whether countrate correction is enabled can be checked with `countrateCorrectionEnabled()`, which is available on the detector and receiver object.

```
d->enableCountrateCorrection();

if (r->countrateCorrectionEnabled()) {
    // ...
}
```

Interpolation

Interpolation of extra large pixels is explained in detail in Section A.1.2.1.

It can be enabled or disabled at runtime with `enableInterpolation()` and `disableInterpolation()` on the detector object. Whether interpolation is enabled can be checked with `interpolationEnabled()`, which is available on the detector and receiver object.

```
d->disableInterpolation();
```

```
if (!r->interpolationEnabled()) {
    // special handling of non-interpolated frames
}
```

Flatfield Correction

If flatfield correction is enabled, the detector counts are integer multiplied with a flatfield correction value. The correction values are measured at specific beam energies and thresholds, and then stored in the module directory as described in Section A.2.6.1. If the actual beam energy, as programmed into the detector, does not match the beam energy for any specific flatfield measurement, then the following algorithm applies to find the correction values:

1. If the beam energy is below any measured flatfield energy, then the flatfield correction values for the lowest beam energy is chosen.

2. If the beam energy is above any measured flatfield energy, then the flatfield correction values for the highest beam energy is chosen.

3. If the beam energy is in between any measured flatfield energy, then the flatfield correction values are interpolated from the values of the two nearest beam energies.

The flatfield for a specific beam energy may be measured at multiple thresholds. If the threshold, as programmed into the detector, does not match the threshold for any specific flatfield measurement, then the correction values from the measurement of the nearest threshold is used.

Users can add their own per module flatfield measurements by extending the flatfield search path in the configuration file:

```
detectors:
  - […]
    modules:
      - […]
        decoding:
          flatfield-path:
            - /path/to/user/flatfields
            - default
```

The `default` entry adds the default flatfields from the module directory to the end of the search path, so that they are still used, if no matching flatfield correction values can be found in the user path. The structure below the user specified flatfield path must have the same structure as defined in Section A.2.6.1.

The flatfield correction data file contains the correction coefficients for each pixel stored in row-major order. Each coefficient is a double-precision floating point number according to IEE754 (8 bytes long), stored in little-endian order. There is no specific requirement to use suffix `.bin`, since filenames are fully configured in the `flatfield.yml` file. However, the suffix used for data files generated by X-Spectrum is `.bin`.

Flatfield correction can be enabled or disabled at runtime with the methods `enableFlatfield()` and `disableFlatfield()` on the detector object. The default is that flatfield correction is disabled. Whether flatfield correction is enabled can be checked with `flatfieldEnabled()`, which is available on the detector and receiver object. The receiver object also provides the `flatfield()` method to retrieve the flatfield correction values, that have been selected based on beam energy and threshold. This method requires to specify, whether the flatfield for the lower or upper threshold shall be returned. There are additional methods to retrieve flatfield metadata such as author and timestamp.

```
    d->enableFlatfield();

    if (r->flatfieldEnabled()) {
        auto ff = r->flatfield(lambda::Threshold::LOWER);
        auto ff_author = r->flatfieldAuthor(lambda::Threshold::LOWER);
        auto ff_ts = r->flatfieldTimestamp(lambda::Threshold::LOWER);
        // ...
    }
```

Compression

Compression cannot be enabled or disabled at runtime. Only the level and shuffle mode can be changed with `setCompressionLevel()` and `setShuffleMode()` on the receiver object. Whether compression is enabled can be checked with `compressionEnabled()` on the receiver object. It also provides additional information such as the compressor name, the level and the shuffle mode.

```
if (r->compressionEnabled()) {
        auto compressor = r->compressor();
        auto lvl = r->compressionLevel()
        auto shuf = r->shuffleMode();
        // process compressed frame
    }
```

A.4.4. Acquisition Parameters

This section gives an overview of acquisition parameters, which can be changed at runtime:

- Operation mode
- Thresholds
- Trigger and Gating mode
- Region of Interest readout

Operation Mode

The initial operation mode of the Lambda detector is defined in the configuration file, as described in Section A.2.4. It is defined as a set of the following information: bit depth, charge summing mode, counter mode, and pixel pitch.

At run time, the operation mode can be changed with one of these methods

```
    d->setOperationMode(lambda::OperationMode(lambda::BitDepth::DEPTH_24));
    d->setBitDepth(lambda::BitDepth::DEPTH_24);
    d->setChargeSumming(lambda::ChargeSumming::OFF);
    d->setCounterMode(lambda::CounterMode::SINGLE);
```

The argument to `setOperationMode()` is a struct of type `xsp::lambda::OperationMode`. The constructor requires at least a bit depth, which is an enum of type `lambda::BitDepth`. The other constructor arguments are optional, if all are given, a call would look like

```
    OperationMode op_mode{
        lambda::BitDepth_12,
        lambda::ChargeSumming::OFF,
        lambda::CounterMode::SINGLE,
        lambda::Pitch::PITCH_55
    };
    d->setOperationMode(op_mode);
```

The other methods are convenient methods to change only a single operation mode setting.

The actual operation mode or part of it is returned by one of the following methods

```
auto op_mode = d->operationMode();
auto bit_depth = d->bitDepth();
auto cs = d->chargeSumming();
auto cm = d->counterMode();
```

Thresholds

Energy thresholds can be set with

```
d->setThresholds(std::vector<double>{6.0});
```

The argument is a vector of up to two values in keV. The number of thresholds to specify depends on the operation mode:

- for single pixel mode, only one threshold needs to be specified
- for charge summing mode or dual counter mode, two thresholds need to be specified, where the first value is the lower threshold and the second value is the upper threshold.

Setting the threshold also sets a default beam energy to twice the value of the threshold, if single pixel mode and single counter mode is selected, and to twice the value of the second threshold, if charge summing mode or dual counter mode is selected. Users may however specify the beam energy separately to a specific values using the `setBeamEnergy()` method. In that case, the beam energy is no longer modified, when thresholds are changed.

```
d->setBeamEnergy(15.0);
```

The actual thresholds and beam energy can be retrieved from the detector object with

```
auto thresholds = d->thresholds();
auto beam_energy = d->beam_energy();
```

On the receiver object the methods are slightly different:

```
auto threshold = r->threshold(xsp::lambda::Threshold::LOWER);
auto beam_energy = r->beam_energy();
```

Triggering

The trigger mode can be set with

```
d->setTriggerMode(xsp::lambda::TrigMode::EXT_SEQUENCE);
```

The argument is an enum of type `xsp::lambda::TrigMode`, which can have the values `SOFTWARE`, `EXT_SEQUENCE`, or `EXT_FRAMES`. If trigger mode is set to `EXT_FRAMES`, the number of frames to take per trigger pulse is defined with

```
d->setTriggerMode(lambda::TrigMode::EXT_FRAMES);
d->setFramesPerTrigger(1);
```

Note

Currently, only 1 frame per trigger is supported.

The actual trigger mode and number of frames per trigger is returned by

```
auto trigger_mode = d->triggerMode();
```

```
auto frames_per_trigger = d->framesPerTrigger();
```

Gating

The gating can be switched on and off with

```
d->setGatingMode(lambda::Gating::ON);
```

The argument is an enum of type `xsp::lambda::Gating`, which can have the values `ON` and `OFF`.

The actual gating mode is returned by

```
auto gating_mode = d->gatingMode();
```

A.4.5. Acquisition Commands

Acquisitions are started and stopped with the methods as described in Section 3.5. In addition to the prerequisite mentioned there, applications for Lambda detectors need to check, whether the following additional prerequisite is fulfilled:

- Sensor High Voltage settled:

  The sensor high voltage is programmed during initialization. Since firmware version 2.2.3 for 750K and 2.3.2 for 250K/60K, the voltage is ramped up in the background, so that the call to `initialize()` returns before the high voltage has reached the configured setpoint. Thus, applications need to call `voltageSettled()` on the `lambda::Detector` object afterwards to determine, whether this is the case.

A.4.6. Readout

Depending on operation mode and sensor configuration, each exposure may create more than one frame. In dual counter mode, two subframes are created, with same frame number but different subframe number. For Hydra type detectors, the number of frames depend on the number of connected discrete sensors.

The methods `numberOfSubFrames()` and `numberOfConnectedSensors()` on the receiver object allow applications to determine how many frames to expect per exposure.

```
auto n_sub = r->numberOfSubFrames();
auto n_conn = r->numberOfConnectedSensors();
auto n_frames_to_expect_per_exposure = n_sub * n_conn;
```

Bibliography

[1] https://www.debian.org

[2] "libxsp - Reference Manual", X-Spectrum, 1.4, Dec 2020

[3] "libxsp - Server Tuning Guide", X-Spectrum, 1.0, Dec 2020

[4] https://yaml.org

[5] "Intel® Ethernet Controller X710/ XL710 and Intel® Ethernet Converged Network Adapter X710/XL710 Family Linux* Performance Tuning Guide", Intel, 1.0, March 2016

[6] https://www.zlib.net

[7] http://www.blosc.org

[8] R. Ballabriga, X. Llopart, „Medipix3RX Manual", CERN, v1.4, 2012