


# Xavia SDK c++ user manual

## Contents

**1 Introduction** **1**

**2 Dependencies** **1**

**3 Installation** **1**
3.1 CMake . . . 2
3.2 Other build systems . . . 2

**4 Deployment** **2**

**5 Usage** **2**
5.1 General code structure . . . 2
5.2 Sensor creation and configuration . . . 3
5.3 Sensor control . . . 4
5.4 Working with point clouds . . . 5
5.5 Errors and Events . . . 8
5.6 System power management . . . 9
5.7 Multi tenant . . . 10
5.8 Examples . . . 10

**6 Development support** **11**
6.1 Testability . . . 11
6.2 Debugging . . . 11

**7 Errors and questions** **12**

**8 User manual for add-on camera support** **12**
8.1 Introduction . . . 12
8.2 Usage . . . 13
8.3 Problems . . . 29

This document is the full, human readable user manual for the Xavia SDK. It is targeted to c++ developers that need to create a system to interact with a Xavia Sensor. Please also examine the examples manual and the API reference. This manual will not explain the point cloud itself and how the data behaves in different sceneries. For those aspects, please read the main 'Xavia user manual'. For information on how to use this SDK in combination with the addon camera, please see the separate camera manual.


- Introduction
- Dependencies
- Installation
  - CMake
  - Other build systems
- Deployment
- Usage
  - General code structure
  - Sensor creation and configuration
  - Sensor control
  - Working with point clouds
  - Errors and Events
  - System power management
  - Multi tenant
  - Examples
- Development support
  - Testability
  - Debugging
- Errors and questions


# 1 Introduction

The Xavia SDK in C++ is created using C++17. It is tested with the g++ (7 and up), cl (MSVC 19.39) and clang compilers on Windows x86_64 (Windows 11 on Intel CPU), Linux x86_64 (ubuntu 24, WSL on Intel CPU) and Linux arm64 (ubuntu 24 on Jetson Xavier NX). The SDK supports CMake integration, but can be used with any build system.

# 2 Dependencies

The runtime SDK is build on the following libraries:

- asio
- plog
- pthread (UNIX only)

The first two are included in the SDK binaries should not create concerns. The pthread library on UNIX should be installed by default on your Linux systems.

If using the SDK on Linux or Windows 64, there is support for the camera subsystem, which has additional dependencies. Please read the manuals of those subsystems for further instructions. If no such manuals are present in your SDK archive, then they don't include support for those subsystems.

# 3 Installation

The installation is also explained into the ReadMe.md file in the source of the SDK download.

Download and unpack the SDK archive for your platform into your development environment.

## 3.1 CMake

Use CMake version 3.20 or newer. Include the SDK by adding the following code to the dependencies section of your CMakeLists.txt file:

```
find_package(Xavia_SDK REQUIRED)
```

Depending on where you unpacked the SDK archive, you might need to supply hints on where to find the SDK. This can be done by defining the variable `set(Xavia_SDK_DIR "<UnpackFolder>/cmake/)`, where `<UnpackFolder>` is the path to where you unpacked the SDK archive, including the archive name.

This will create a number of targets, the most important one being `xavia::SDK`. Link this target to the relevant executable or library in your code to make the SDK available:

```
target_link_libraries(MyLib PRIVATE xavia::SDK)
```

In the files of that library you will then be able to include the public header files of the SDK. All of these are in a subfolder `xavia`. Hence the include command is:

```
#include <xavia/iSensor.h>
```

## 3.2 Other build systems

When not using CMake or (versions before 3), you will need to manually include the headers and link the libraries. These files can be all be found in the root of the archive:

- `<root>/include` contains the folder `xavia` and underneath that the public header files. It is advised to setup the system such that the `xavia` folder needs to be mentioned in the include statement: `#include <xavia/iSensor.h>`.
- `<root>/lib` contains the static libraries (.lib and .a) for inclusion during the build process, and shared libraries for Linux (.so) for runtime inclusion.
- `<root>/bin` contains the shared libraries for Windows (.dll) for runtime inclusion.

On unix you will need to link to pthread in order to build.

# 4 Deployment

The SDK supports both shared and static inclusion. When including statically, no extra files are required during deployment. For dynamic inclusion, the file `Xavia_SDK.dll` or `Xavia_SDK.so` needs to be bundled with the executable.

It is not necessary to deploy any of the other files in the SDK archive to the target device for executing your application.

# 5 Usage

## 5.1 General code structure

The SDK partially acts as a framework and usage is split into two parts.

In the first part you should configure the system and the framework. The default configuration already allows to simply start and stop the sensor. By providing callback methods to the factory, the SDK will call these callbacks on specific events to allow you to interact with the system. This part of the SDK works on the rhythm of the hardware. You are in this configuration part for as long as you are interacting with the factory.

In the second part, you control the sensor state and you can move the sensor from 'idle' to 'running' and back. Additionally, it is possible to query the sensor for its state and relevant

properties. This second part starts from the moment you acquire a `Sensor` object by calling the `SensorFactory::Build()` method.

In code this can be recognized as follows:

```
int main()
{
  xavia::SDK::SensorFactory factory;
  {
    // PART 1: CONFIGURATION
  }

  auto sensor = factory.Build();
  {
    // PART 2: RUNTIME
  }
}
```

It is possible to switch as often between these parts as desired. It is also possible to reuse the same factory to create multiple sensors objects.

> ! Warning. Although it is possible to create multiple sensor objects that connect to the same Xavia sensor at the same time, this is not advised. Each such object will take up further resources and requires a separate communication channel, slowing down the overall performance. Within the same application, create only a single sensor object and use a `shared_ptr` to distribute it.

### 5.1.1 Destruction

All the objects used and returned by the SDK are smart pointers, values or references. No raw pointers are used. All these objects are also following RAII principles. The consequence is that you do not need to worry about any of the memory allocated by the SDK or the objects provided by it. Simply allow all to go out of scope for automatic cleanup.

## 5.2 Sensor creation and configuration

In the 'configuration' part of your application, you should setup the interaction between your code and the SDK. This is done through callbacks. In this sense, the SDK acts as a framework that runs its own thread and calls your code from that thread. This thread is managers the TCP connection to the Xavia sensor.

The sensor factory provides the following callback options:

- `SetPointCloudCallback`
- `SetErrorCallback`
- `SetAlertCallback`

The actual response you will get on each of these callbacks is explained in separate sections later in this manual.

When all configuration has been completed, you can instantiate the `Sensor` object using the `Build` method. This method has 2 optional arguments: the ip address of Xavia and the port number. The following variations are all legal:

```
factory.Build();
factory.Build("10.10.100.11");
factory.Build("10.10.100.11", 4000);
```

Since these are the default values, all three will have the same effect. The `Build` function creates the actual connection to the Xavia sensor. As such it will only succeed if the sensor is detectable in

the network and powered on. In the case it is not, the function might take several seconds (up to 15 seconds) before reporting the error. Errors are reported by throwing exceptions.

## 5.3 Sensor control

In the 'runtime' part of your application you can control the Xavia sensor by calling methods on the `ISensor` object:

- `Start`
- `Stop`
- `GetSerialNumber`
- `GetCurrentState`
- `GetErrors`
- `Reboot`
- `Poweroff`

All of these are synchronous operations that will execute either locally or remotely. Synchronous in this sense means that the method call will only return after either a timeout or a response from the sensor. For functions like `Start`,`Stop`, `Reboot` and `Poweroff` this response only indicates that the sensor accepted the command into the queue but it is probably not yet executed. The `GetSerialNumber` and `GetCurrentState` will be discussed below, but do not directly interact with the sensor. Instead they serve information from the SDK cache. The `GetErrors` function does synchronize with the hardware and retrieves the latest information.

### 5.3.1 Start/Stop

The start action enables the laser output, starts the measurement and starts streaming the point clouds from the sensor to all clients. The stop action does the inverse action: stops laser output, stops measurement and stops streaming point clouds. Both actions will block your thread until either a timeout occurs, or the sensor acknowledged receiving the command. Therefor it is good practice to encompass both in a try catch block:

```
try
{
  sensor->Start(); // or Stop()
}
catch(xavia::sdk::Exception& e)
{
  //..
}
```

The function has an optional `timeoutMs` argument. You can provide the maximum amount of time in milliseconds the SDK waits for the response from the sensor. If the timeout is omitted the default value of 5 seconds is used. Timeouts can happen on `Stop` if the sensor is also streaming other messages at 1kHz and the outgoing buffer was full when the message was supposed to be transmitted. In this case it is safest to synchronize the state to make certain that the stop really failed:

```
try
{
  sensor->Stop();
}
catch(xavia::sdk::Exception& e)
{
  case UNABLE_TO_STOP:
    sensor->GetErrors(); // synchronizes
    if(sensor->GetCurrentState() == xavia::sdk::SensorState::IDLE)
```

```
    {
      // stop worked, the sensor is idle
      break
    }
    else
    {
      // Error failed to stop
    }
    break;
}
```

The errors `ACTION_ILLEGAL_SENSOR_RUNNING` and `ACTION_ILLEGAL_NOT_RUNNING` will be returned if the sensor was already started or stopped respectively. This means that calling `Start` on a sensor that is already running is considered an error. But the error will not compromise the sensor. It will refuse the command and continue in state it was.

(i) Note: All clients are considered equal by Xavia. When multiple clients connect, and one sends the Stop command, streaming will stop for all clients. This can happen for example when using the web viewer together with your own application. See the multi-tenant section.

### 5.3.2 Sensor state

Xavia has 4 states:

- **Uninitialized**: This state can only occur if there is an internal error in the hardware which prevents the on board software from communicating with the certain components. Normally this case should never occur.
- **Idle**: The default startup state. The sensor is ready and waiting for the `Start` command
- **Running**: The sensor is measuring, the lasers are on.
- **Error**: The sensor experienced an error and is unable to operate normally. Advised action is to reboot the sensor.

The idle and running states will be encountered most often. Use the `Start` and `Stop` functions to switch between them.

The error state can happen for a number of reasons: an internal error, overheating, under voltage, ... When the system is in an error state, it will remain that way. The only recovery from an error state is by rebooting the system.

## 5.4 Working with point clouds

Once the sensor is in running state it will stream point clouds to all connected clients. The SDK will automatically receive all these point clouds and handle them. To receive these point cloud objects, a point cloud callback should be registered. To do so, use the `SetPointCloudCallback` function of the `SensorFactory` before building the Sensor object:

```
factory.SetPointCloudCallback(callback);
```

This callback has only one argument and no return. Additionally it should not throw exceptions. For example:

```
void MyCallback(std::unique_ptr<xavia::sdk::IPointCloud> pc)
{
  // do something with pc
};
```

You receive the point clouds as a mutable, unique pointer. As such their lifetime is now connected to the scope of this function unless you decide otherwise.

### 5.4.1 Point cloud data

A few explanation of the point cloud data and the meaning of different fields is part of the full product user manual. Here only a short summary is provided, focused more on the technical side then the physical interpretation of the data. The point cloud object provides access to the following fields:

- `GetXMm()`, `GetYMm()`, `GetZMm()`: returns the reconstructed (x,y,z) coordinates expressed in mm with the optical origin of the sensor.
- `GetDistanceMm()`: provides the radial distance from the sensor origin to the point in mm. This is calculated from the time of flight information. This field is always larger than 0. If this field is -1, then no valid time of flight measurement was received.
- `GetReflectivity()`: a distance corrected measure for the amount of laser light that returns. It is a number larger than 0. If it says 10%, it means that the returned light is equal to the return of a 10% Lambertian target at the measured distance.
- `GetAmbient()`: provides a measurement of the background light. This means IR light from other sources that the lidar illuminated the scene.
- `GetConfidence()`: returns the confidence of the measurement. This is a value between 0 and 1 where 1 would mean 100% confident. You should expect to see values around 0.995 and 0.999 in this field.

For the (x,y,z) coordinates, the coordinate system is a right-handed system, with the x-direction equal to the viewing direction of the sensor. The z-direction is upwards (when the XenomatiX logo is on top) and the y-direction is to the left hand side.

The point cloud information is communicated line per line. The standard Xavia has a point cloud frame of 56 lines, and each line provides 192 measurements. Lines can be identified from the `MetaData::lineNumber` field. The vectors returned by e.g. `GetDistanceMm()` contain the distance value for all 192 measurements in this line. For future compatibility the number of measurements in a line can be read from either `GetDistanceMm().size()`, or from `GetNrPoints()`.

56 lines together cover the entire field of view. That combination is called a 'frame'. Frames can be identified by the `MetaData::frameNumber` field. It starts at 0 and increments for every frame that is recorded by the hardware. The maximum line number is 4'294'967'295, after which it will overflow back to 0. This will happen after 2485 days of continuous measurement (6,8 years) for the 20Hz version.

An important note is that within a frame, the lines are reported from line number 0 to 55. This is from the bottom towards the top of the field of view.

**5.4.1.1 Metadata** Besides the point cloud data itself, the point cloud object also provides metadata. Metadata are counters and timestamps related to the point cloud information.

Timestamps are uint64_t POSIX numbers indicating the number of microseconds since 1/1/1970. The line-timestamp reported is the internal clock of Xavia at the moment of the exposure of that specific line. The frame-timestamp is equal to the line timestamp of the first exposed line (line 0) of the frame. There is around 893 us between each line.

The metadata also reports a return index and a trigger counter. The return index will be used in the future to indicate multiple returns. When this becomes active, the sensor will send multiple lines with the same line number, but different return index. The meaning of each index will depend on the selected mode. Each such line will trigger a new callback.

The trigger counter will be active when the sensor is put into slave mode. In this case, some external signal will be used to start the exposure of the sensor. If this signal is lower than the maximum framerate of the sensor, then the trigger counter will equal the frame number. However, if triggering happens faster than the sensor can perform measurements, then the trigger counter will grow faster than the frame number. In other words, if the frame number increments by 1, but the trigger counter increments by 2, it means that the system received a trigger but was not ready to act on it

and that trigger did not result in a measurement. On the other hand, if both the trigger counter and frame number increment by 2, it means that the trigger did result in a measurement, but the system was unable to stream the data or the sdk failed to receive it properly.

**5.4.1.2 Probes** The point cloud object also contains a `OnboardProbes` section that gives access to a temperature and a humidity readout. Both are measured from hardware probes inside the sensor. The temperature probe is reported in degrees Celsius. The humidity is reported as the relative humidity.

#### 5.4.1.3 IMU data

> ! Note: the IMU is not supported in the first version of Xavia.

Each point cloud also carries the IMU data. The IMU is part of Xavia and measures the acceleration and angular motion of the device. This can be used to either correct for motion blur, better synchronize with the RGB image, or to help slam algorithms. It will provide the acceleration on the three axes expressed in m/sˆ2. It will also provide the angular motion for yaw, pitch and roll expressed in degrees per second.

### 5.4.2 Performance

To use the Xavia to its fullest, some performance considerations are in order.

**5.4.2.1 Real time processing** It is important to note that he Xavia works at a framerate of either 20Hz or 10Hz, depending on the options your purchased. At 20Hz and 56 lines per frame, the sensor will output 1120 lines per second. This will also result in your callback being called 1120 times per second. In order to not miss any data, the callback should be handled as a real time part of the software. Meaning that it should complete its work within 893 microseconds. If the callback takes more time than that, then eventually the internal buffers will fill up and data will be skipped.

In the 10Hz version, the 56 lines of a frame will still arrive every 893 microseconds, but a 50 ms pause will be present before the next burst of data comes.

The choice for streaming line by line instead of aggregating on the sensor and streaming full frames stems from the idea that some data is better than no data.

- Your code can already start interpreting the first lines before the final lines are on the system. This can save milliseconds in response time. For example, the bottom half of the FoV (which on a vehicle typically contains the ground and closest objects) can already be processed 25 ms in advance of the full frame arriving.
- Even if a single line is missing, the other 98% of the frame can still be enough to draw conclusions and run your algorithms.

The SDK has an internal buffer to help manage variations in the network communication or in the callback response. The default size of this buffer equals two full frames. This should be less than 0.5MB of ram. The size of this buffer is configurable through `SensorFactory::SetPointCloudBufferSize(size)`. Lowering this value can save ram, at the danger of missing lines. We don't advise to go below 3 or many lines will be missed. Any value above 20 should be sufficient to capture all lines.

> ! Warning: a large buffer is no long-term solution for a slow callback. If the callback is too slow, the buffer will prevent lines to be dropped, but the data provided to the callback will become increasingly stale. After a while, the buffer will still overflow and lines will be missed.

The advise is to use the callback only to move the data into a new thread to do the calculations. In this thread either aggregate the lines into a frame if that is the desired methodology, or use a

multi-threaded computation pipeline to process multiple lines at once, keeping each step below the 893 microseconds.

**5.4.2.2 Vectorization** A feature of this SDK is that the data is provided in a vectorized manner. This shows in return value of the `GetXMm()` methods and its friends. The `std::vector` container provides the data in consecutive memory. This allows user to optimally use `std::memcpy` to transfer data, but it also allows the compiler/user to optimally use caching and simd operations. This also supports the memory organization that many blas libraries prefer.

### 5.4.3 Exporting and importing data

With access to data from the point cloud it is possible to support any output format. However, to ease development, this SDK provides the proprietary XenomatiX format. For this, two functions exist: `Serialize()` and `Deserialize()`. Their usage can be explained by the following example:

```
void OnNewPointCloud(xavia::sdk::IPointCloud& pc)
{
  auto buffer = pc.Serialize(); // returns the XenomatiX format
  std::ofstream file("output.xpc", std::ios::binary);
  file.write(buffer.data(), buffer.size());
}

std::unique_ptr<xavia::sdk::IPointCloud> ReadPointCloud(std::string filename)
{
  auto pc = xavia::sdk::CreatePointCloud(); // creates empty point cloud

  // read from file
  std::ifstream file(filename,std::ios::binary| std::ios::ate);
  std::streamsize size = file.tellg();
  std::vector<char> buffer(size);
  file.read(buffer.data(), size);

  // interpret
  pc.Deserialize(buffer); // pc will be equal to the pc from the 'OnNewPointCloud' function.
  return pc;
}
```

The serialized format is equal to the ethernet communication, as such it is also possible to use the `Deserialize` function to interpret the bytes from e.g. a network capture.

## 5.5 Errors and Events

Errors can occur in two scenarios: 1) you asked for an action and it is unable to comply, 2) something happened in the system and it reports the error. The first category is triggered by a user action. They are all communicated by throwing exceptions. This is a non-exhaustive list of possible errors:

- **UNABLE_TO_CONNECT**: SDK failed to connect to target IP address. Check your network connection and that the system is powered.
- **ACTION_ILLEGAL_SENSOR_RUNNING**: You cannot perform this action while the sensor is in running state (e.g. Start)
- **ACTION_ILLEGAL_SENSOR_NOT_INITIALISED**: You cannot perform this action while the sensor is not initialized (e.g. Start)
- **UNABLE_TO_START**: Sensor failed to acknowledge the start command
- **UNABLE_TO_STOP**: Sensor failed to acknowledge the stop command
- **ACTION_ILLEGAL_NOT_RUNNING**: You cannot perform the action while the sensor is in idle state (e.g. Stop)

- **COMMAND_BUFFER_FULL**: The command was received by the system, but the current command buffer is full. Try again later.
- **UNKNOWN_MESSAGE_ID**: This system does not recognize the command you provided.
- **UNSUPPORTED_COMMAND**: The provided command is not supported by the current system.
- **UNKNOWN_ERROR**: Error that does not fall in any of the above categories. Use the `.what()` method of the exception or the logs to get more information.

Some extra explanation on some of these errors:

- Command buffer full: The Xavia has an internal queue of 6 commands. If more commands are send while the previous command are still being processed, it will report that the buffer is full. In this case your command is not accepted and the system will not execute it. Try to send it again when the buffer has room.
- UNKNOWN_MESSAGE_ID and UNSUPPORTED_COMMAND will be very rare. These errors can only occur when mixing an old sensor with a newer SDK. Or if you try to use the SDK for an unsupported XenomatiX sensor.

Besides the errors above, it is also possible that something happens on the system and an error is pushed. In this case the error is not caused by the call to a function and the sdk cannot throw an exception. These errors are pushed to the Event callback if you provide any (not obligatory). The following events are currently defined:

- **HARDWARE_CRITICAL**: An unspecified critical hardware error occurred. Usually this means a broken component.
- **POWER_CRITICAL**: The measured power on the main input, or in one of the components was out of bounds. The sensor will shut down to protect itself.
- **TEMPERATURE_CRITICAL**: The measured temperature is out of range. This either indicates that the sensor is overheating, under cooled, or that a temperature sensor is broken. In all cases, the sensor will take actions to protect itself and shut down.
- **EYE_SAFETY_ALERT**: The sensor measured an unexpected and potentially unsafe laser output and prevented laser output before it can do any harm. The sensor will stay in this error state until rebooted.
- **HUMIDITY_ALERT**: The measured humidity inside the sensor is out of bounds and the sensor was stopped to protect itself.
- **VOLTAGE_ALERT**: An invalid voltage was measured. Correct the input voltage you provide and make sure that your power system can support the load.
- **HOUSING_REMOVED**: This event indicates that the sensor is or was opened. To protect users certain components and laser output have been disabled. Replace the housing and restart the sensor to resolve the warning.

These events (sometimes also called Alerts) are communicated to the callback together with a `state` flag. If this flag is true, it means that the alert is raised. If the flag is false, then the alert is released. For example: if the sensor overheats the `temperature_critical` alert is issued with status true. If after a while the sensor cools down, the alert will get triggered again but with the status false.

## 5.6 System power management

The SDK allows to power off and reboot the Xavia sensor. In order to do so, you first need to create a connection with the system. Once the `Sensor` object has been build and connection was successful, the `Poweroff` and `Reboot` commands become available. In code this looks like:

```
xavia::SDK::SensorFactory factory;
auto sensor = factory.Build();
sensor->Reboot(); // or sensor->Poweroff();
```

In both cases, the SDK will issue the command to the sensor over the TCP/IP connection. Assuming the sensor is responsive and has room in the command queue, it will respond with the message 'command queued'. The SDK will consider this success and the call to `sensor->Reboot()` will return. At this point the command on the Xavia sensor is simply queued. Depending on the queue and the other commands, it might take some time before the command is actually executed. When the Xavia executes the command to shutdown it will signal the operating system to stop all activities and gracefully shutdown. At this point the Xavia sensor will broadcast the message that it is 'shutting down' to all clients. If you connected an event callback (see the Errors and events section), this callback will get triggered on the `xavia::SDK::Alert::shutting_down` event.

Due to the asynchronous nature of the commands, this can generate different behavior on different execution runs. Therefore, the safest action is to assume the system will go down and consider the sensor object to become invalid from the moment you use either the `Poweroff` or `Reboot` method. Destroy the sensor object (let it go out of scope, or use `sensor.reset()`). When rebooting, wait some time for the hardware to reboot and try to use the factory to build a new sensor object. The emphasis is on 'try' because this might fail if the system is not yet ready to accept client connections.

In the case the sensor does not react positively on the TCP command, both `Poweroff` and `Reboot` methods will throw an exception. In this case you can assume that the sensor is not powering down.

## 5.7 Multi tenant

The Xavia system is a multi tenant system, which means that multiple clients connect to the same sensor instance. None of these clients are considered more important than the other. The sensor will handle each received command in the order they arrive. The consequence is that if any of the clients issues e.g. a `Start` command, the sensor will obey. For all important scenarios (start, stop, shutting down, ...) the sensor will broadcast an event alerting all the clients of the state change. If needed you can query the system to synchronize with its state by using the `GetErrors()` function (see the section on sensor state).

Additionally, the SDK will not take automatic actions. For example: destroying the `Sensor` object will not send a Stop command to the Xavia sensor. It will keep on working, even if no clients are listening. Similarly, the SDK will not issue a Start command when connecting. Instead

You can use this, if so desired, in a micro services like architecture where different clients act on different signals, or take different actions. E.g. one client can handle all the point clouds and events, while another system can create a temporary client just to start and stop the sensor(s). Or a central system can be used to create a sensor object simply to stop and shut down all Xavia sensor in your network. Each individual application will then receive the shut down signal from their own sensor.

Another example is the interaction with the build-in Point cloud viewer. This viewer also offers start, stop, power off, and reboot buttons. You can use this:

- to check if your own Start and Stop commands truly change the system state correctly.
- to check if your application can handle the sensor starting/stopping based on alternative sources.

### 5.7.1 Security

The current version of the sensor and SDK do not contain any security measures. Messages are not encrypted during transmission. No authentication is required for commanding the system.

## 5.8 Examples

To further support your development, the SDK contains a number of examples. These examples can be used as a starting point for your code or to act as an implementation example. All the examples use and link the SDK in the same way your application should use it. Each example focusses on a

specific functionality and is provided with detailed comments. The ideal application combines the functionalities of the individual examples into a robust system.

The examples can be built easily using cmake and will then be available as a target. Either include the `examples` subfolder into your project or build it directly:

```
cd Xavia_SDK-etc.
cmake -S ./examples -B ./examples/build -G {"Visual Studio 17 2022", "Ninja", etc.}
cmake --build ./examples/build --target xavia::example_basic
```

The above shows how to build the `xavia::example_basic` target. This will create an executable `xSDK_example_basic`. More example targets are available, for the full list, please review the examples manual.

# 6 Development support

As fellow developers we understand that creating an application is more than just writing code and getting it to work. Professional software development requires thinking about code quality, testability and readability as well. This SDK has been designed with your quality of life in mind, as will be explained in the following paragraphs.

## 6.1 Testability

In order for you to achieve 99.99% coverage, we provide all the interfaces for our objects. This means that it is possible to mock the sensor and the point clouds.

For example, using the gmock framework from google:

```
#include <xavia/iSensor.h>
#include <gmock/gmock.h>

class MockSensor : public xavia::SDK::ISensor {
 public:
   MOCK_METHOD(void, Start, (const std::size_t& timeoutMS), (override));
   MOCK_METHOD(void, Stop, (const std::size_t& timeoutMS), (override));
   MOCK_METHOD(SerialNumber, GetSerialNumber, (), (const, override));
   MOCK_METHOD(SensorState,  GetCurrentState, (), (const, override));
   MOCK_METHOD(std::vector<SensorError>, GetErrors, (const std::size_t& timeoutMS), (override));
   MOCK_METHOD(void, Reboot,(), (override));
   MOCK_METHOD(void, Poweroff,(), (override));
}
```

creates a fully controllable mock xavia sensor on which you can define expectations and put behavior. In a similar way the full point cloud can be mocked.

The combination of both should allow you to write unit tests for most of your code.

## 6.2 Debugging

Despite all the testing, bugs can still happen. It is not possible for you to step into our code during debugging due to the pre-built nature of the SDK. However, to aid understanding in what is going wrong, the SDK can output logs. Two options are available:

- `xavia::SDK::SetLogOutputFolder()` provides a way to designate a folder in which the SDK will create a file. Each execution a new file will be defined using the system time.
- `xavia::SDK::SetLogCallBack()` provides a way to capture the SDK log message into your own application. The callback will receive a `severity` and `std::wstring` field. This allows you to push these log files to your own file or e.g. send them to a cloud or central system.

If neither of these are provided, the SDK will log to the `std::cout` by default. It is possible to provide both a folder/cout output and the callback method.

For more information on the logging, please examine the corresponding example.

> ! Important: Set the preferred logging method before calling any SDK function. On the first log message, the logging instance is initialized and the logging method can no longer be changed after that.

# 7 Errors and questions

If, after reading this manual and the examples, you still cannot find the answer to your questions, please contact us at xavia.support@xenomatix.com.

# 8 User manual for add-on camera support

This document is the comprehensive guide for using the camera subsystem of the Xavia SDK. It covers camera initialization, configuration, image acquisition, data formats, and error handling. For general SDK usage patterns and lidar sensor information, please see the Xavia SDK user manual.

- Introduction
  - Features
  - Requirements
  - Dependencies
  - Integration
- Usage
  - Camera object management
  - Data management
  - Offline features
  - Error handling
  - Single frame acquisition
- Problems
  - Symbolic links

## 8.1 Introduction

The Xavia SDK provides control for the camera add-on to the lidar sensor. The camera subsystem is independent from the lidar system, allowing simultaneous acquisition of synchronized lidar point clouds and camera images. For combined usage, see Connection with lidar object.

---

### 8.1.1 Features

- **Camera control**: Start, stop, and monitor camera streaming.
- **Multiple color formats**: RGB8, RGB12, BayerRG12 (packed and unpacked), and BayerRG8.
- **JPEG storage**: Save images as 8-bit JPEG files with configurable quality and compression.
- **Exposure control**: Manually adjust exposure time.
- **Factory reset**: Reset all camera parameters to factory defaults for troubleshooting.
- **Full metadata**: Access frame numbers, timestamps, and pixel data for custom processing.
- **Offline serialization**: Save and load image bytes with full 12-bit quality.

### 8.1.2 Requirements

**Supported platforms**: Windows x86_64 and Linux x86_64.

**Hardware**:

- Gigabit network connection with jumbo frames enabled
- Camera must be powered and reachable on the configured IP address
- Sufficient processing power for JPEG encoding if saving frames in real-time

### 8.1.3 Dependencies

The Following libraries are used in the camera subsystem of the XaviaSDK:

- Linux only: `libibverbs1` and `librdmacm1`

**Linux packages**: If running on Linux, install the following packages:

```
sudo apt-get -y install libibverbs1 librdmacm1
```

### 8.1.4 Integration

There are no special installation instructions. Install the Xavia SDK as described in the user manual.

---

## 8.2 Usage

### 8.2.1 Camera object management

The camera follows the same configuration and runtime pattern as the lidar sensor. Configure the camera using `SensorFactory`, then control it at runtime via the `ICamera` interface. It will provide objects of the `IImage` interface.

**8.2.1.1 Create the camera object** To use the camera, you must first register an image callback with the factory. This callback is required to activate the camera subsystem. The callback will be invoked each time a frame is ready for processing.

```
#include <xavia/sensorFactory.h>
#include <xavia/iSensor.h>
#include <xavia/iCamera.h>
#include <xavia/iImage.h>

void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  // Process incoming image (see section on data management)
}

int main()
{
  // PART 1: CONFIGURATION
  xavia::sdk::SensorFactory factory;

  // Register image callback (required to activate camera)
  factory.SetImageCallback(imageCallback);

  // Optionally configure camera settings
  factory.SetImageOutputFormat(xavia::sdk::PixelFormat::RGB8);
  factory.SetCameraIPAddress(L"10.10.100.12");

  // PART 2: RUNTIME
  auto sensor = factory.Build();
```

```
  // Get camera from sensor
  auto camera = sensor->GetCamera();

  // Always verify camera is available
  if (!camera)
  {
    std::wcerr << L"Camera not available" << std::endl;
    return EXIT_FAILURE;
  }

  std::wcout << L"Connected to camera: " << camera->GetCameraIdW() << std::endl;
  return EXIT_SUCCESS;
}
```

If no callback for the camera is provided, the SDK will skip searching for the camera. In that case, or if the camera cannot be found, the function `GetCamera()` will return a `nullptr`.

> Note: the camera object is provided as a shared pointer. This means that it will stay in existence for as long as you hold on to that pointer. Even if the sensor and sensorfactory objects are destroyed. However, there is no need to manually destroy it. It is a smart pointer. If you do loose the pointer, you can always reclaim it from the `GetCamera()` function since the sensor object also holds on to a shared pointer. The camera will be automatically destroyed once you released all your versions of the shared pointer AND the sensor object is destroyed.

The example demonstrates the call to the `SetCameraIPAddress()` function. This is optional. Omitting that call will make the SDK use the default camera IP address "10.10.100.12".

The `SetImageOutputFormat()` call is also optional. See the section on image formats for more information.

#### 8.2.1.2 Start and stop the camera

Once you have the `ICamera` object, you can control the camera streaming.

**Starting the camera**:

The `Start()` method begins streaming images from the camera. Images are delivered to your callback registered with `SetImageCallback()`. If the camera fails to start within the timeout period, an exception is thrown. The default timeout is 5000 milliseconds. You can override it with a custom value:

```
try
{
  camera->Start();  // Default 5-second timeout
  // or with custom timeout:
  // camera->Start(10000);  // 10-second timeout
}
catch (const xavia::sdk::Exception& e)
{
  std::wcerr << L"Failed to start camera: " << e.what() << std::endl;
}
```

**Stopping the camera**:

The `Stop()` method halts the image stream. Once stopped, no further callbacks will be invoked until `Start()` is called again:

```
try
{
  camera->Stop(); // default 5 seconds timeout
}
catch (const xavia::sdk::Exception& e)
{
  std::wcerr << L"Failed to stop camera: " << e.what() << std::endl;
}
```

Similar to the `Start()` function, you can provide your own custom timeout in which you expect a response to the stop command.

**Checking streaming state**:

Use `IsStreaming()` to query whether the camera is currently streaming:

```
if (camera->IsStreaming())
{
  std::wcout << L"Camera is actively streaming" << std::endl;
}
```

Streaming is true between start and stop. Calling start on a camera that is already streaming is considered an error and will throw an exception. If there is any risk on this, use the `IsStreaming()` to check the state of the sensor before attempting state changes.

#### 8.2.1.3 Connection with lidar object

The camera and lidar sensor are independent subsystems and can be controlled separately. However, they can operate simultaneously and in synchronization.

Both the camera and lidar use the same `SensorFactory` for configuration:

```
xavia::sdk::SensorFactory factory;

// Configure both systems
factory.SetPointCloudCallback(pointCloudCallback);
factory.SetImageCallback(imageCallback);

auto sensor = factory.Build();

// Get both cameras and lidar
auto camera = sensor->GetCamera();

// Control independently
if (camera) camera->Start();
sensor->Start();

// both callbacks will get triggered.

sensor->Stop();
if (camera) camera->Stop();
```

For examples of simultaneous camera and lidar usage, see the `combinedLidarCamera` example in the examples folder. Each frame carries metadata including timestamps, allowing you to synchronize camera and lidar data post-processing if needed.

> Note: Not all versions of Xavia support time synchronization using ntp or ptp. If no time synchronization is set up, lidar and camera will use different clocks and the timestamps cannot be sensibly compared. In that case we advise to use a visual cue and do manual time synchronization.

The add-on camera and the Xavia lidar are synchronized using hardware triggers. As a result it is guaranteed that the exposure of each camera frame coincides with the exposure of a lidar frame. (partially because they will have different exposure times). Additionally, the lidar is setup to trigger the camera at 20 or 10Hz (depending on the Xavia type), even if the lidar itself is not capturing data. This allows you to use the camera independent of the lidar.

For applications like coloring the point cloud, or when the point cloud is the main object of attention, we advise to first start the camera and then the lidar. At the end, first stop the lidar and then the camera. This ensures that there are camera frames for every point cloud.

#### 8.2.1.4 Configure exposure time

Camera exposure time controls how long the sensor integrates light for each frame. This directly affects image brightness and sensitivity to motion blur. By default the camera is set to auto exposure with limits to guarantee that it can keep up with the 20 Hz of the lidar. However, you can set the exposure time to a fixed value at any time during the runtime phase.

Use `SetExposureTimeUs()` to set exposure time in microseconds:

```
// Set exposure time to 10 milliseconds
camera->SetExposureTimeUs(10000);
```

The exposure time is limited in range, depending on the settings of the camera. The `SetExposureTimeUs()` function will attempt to set whatever value you provide, but the camera will limit it to an allowed value closest to your requested value. If the requested value does not match the actual value, the SDK will continue, but will issue a warning in the log.

The SDK will automatically make sure that the exposure time of the camera lines up optimally with the exposure time of the lidar. For example, if you request a camera exposure of 10ms, but the lidar exposure time is 50ms, then the SDK will also apply a 20 ms delay on the camera trigger such that the camera frame exposure is in the middle of the lidar frame exposure.

Setting the exposure `SetExposureTimeUs(0)` to 0 us, will return the sensor to auto exposure.

**Querying current exposure**:

```
uint32_t currentExposureUs = camera->GetExposureTimeUs();
std::wcout << L"Current exposure: " << currentExposureUs << L" us" << std::endl;
```

This function will query the current exposure time from the camera.

#### 8.2.1.5 Reset configuration

If the camera is not behaving as expected or you want to restore factory defaults, use `ResetConfiguration()`:

```
camera->ResetConfiguration();
```

This method resets all camera parameters to factory defaults, including:

- Exposure time
- Gain settings
- Triggering mode
- Frame rate
- Pixel format

  **Warning**: `ResetConfiguration()` will overwrite your current settings. Use only when you intend to restore defaults.

Alternatively, if you did not change the user settings of the camera through the vendor software, then simply restarting the camera will also reset the settings to the factory defaults.

When connecting to a camera, the SDK will perform a sanity check of the settings of the camera. If these do not match the expected settings, a warning will be logged, but the SDK will continue

with the provided settings. This will allow you to change any setting using the vendors ArenaView software and still use the Xavia SDK.

**8.2.1.6 Camera privileges** Contrary to the sensor, the camera follows a single master philosophy. Only the first client that connects to the camera has write access and can change settings or state. If, for example, ArenaView is connected to the camera, then the SDK will not have write access to the camera. In that case the SDK usage is very limited:

- It cannot start and stop the camera.
- It cannot retrieve the rgb data.
- It cannot set the exposure time or any configuration.

It is possible to check the camera privileges using the `HasWriteAccess()` query. It will attempt to write one of the features and return true on success or false on failure.

If you attempt to continue without write access, you will see genicam errors appearing that say "no access" or "not allowed" or a variation thereof.

**8.2.1.7 Other Queries** The camera object also provides the following queries:

- `GetCameraId()`
- `GetCameraIdW()`

Both return the camera identification number (serial number).

---

### 8.2.2 Data management

**8.2.2.1 How to get RGB frames** Images are delivered to your callback via the `SetImageCallback()` method registered on the factory. Each callback receives a complete `IImage` object containing pixel data and metadata.

**Image callback signature**:

```
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  // Process image here
}

factory.SetImageCallback(imageCallback);
```

**Accessing image metadata**:

Use `GetMetaData()` to access frame information:

```
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  const auto& metadata = image->GetMetaData();

  uint32_t frameNumber = metadata.frameNumber;       // Sequential frame counter
  uint64_t timestampUs = metadata.frameTimeStampUs; // Timestamp in microseconds
  uint32_t width = metadata.width;                   // Image width in pixels
  uint32_t height = metadata.height;                 // Image height in pixels
  xavia::sdk::PixelFormat format = metadata.pixelFormat;  // Current pixel format
}
```

The frame timestamp is provided by the camera, in its internal clock and represents the start time of the exposure.

**Accessing pixel data**:

Use `GetPixelData()` to access raw pixel values. The format and size depend on the configured pixel format:

```
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  const auto& metadata = image->GetMetaData();
  const auto& pixelData = image->GetPixelData();
  const auto pixelFormat = metadata.pixelFormat;

  size_t totalBytes = pixelData.size();
  size_t bytesPerPixel = totalBytes / (metadata.width * metadata.height);

  // Access pixel at coordinates (x, y)
  uint32_t x = 100, y = 200;
  auto pixelPtr = pixelData.data() + ((metadata.width * y + x) * bytesPerPixel);

  switch(pixelFormat)
  {
    case: xavia::sdk::PixelFormat::RGB8:
      // RGB8 format: 3 bytes per pixel
      uint8_t r = pixelPtr[0], g = pixelPtr[1], b = pixelPtr[2];
      break;
    case xavia::sdk::PixelFormat::RGB12:
      // RGB12 format: 6 bytes per pixel (3 uint16_t)
      auto pPx16 = reinterpret_cast<const uint16_t*>(pixelPtr);
      uint16_t r = pPx16[0], g = pPx16[1], b = pPx16[2];
      break;
    case xavia::sdk::PixelFormat::BayerRG12:
      // BayerRG12 format: 2 bytes per pixel (1 uint16_t)
      auto pPx16 = reinterpret_cast<const uint16_t*>(pixelPtr);
      uint16_t v = pPx16[0];
      break;
    case xavia::sdk::PixelFormat::BayerRG8:
      // BayerRG12 format: 1 byte per pixel
      uint8_t v = pixelPtr[0];
      break;
    case xavia::sdk::PixelFormat::BayerRG12packed:
      // much more complicated
  }
}
```

For more information on interpreting the image data see the section on image formats.

**8.2.2.2 Image formats** The camera supports multiple pixel formats to balance quality, bandwidth, and processing requirements. Configure the format during the configuration phase, before calling `Build()`:

```
factory.SetImageOutputFormat(xavia::sdk::PixelFormat::RGB8);
```

**Important**: The pixel format cannot be changed after `Build()` is called. Choose your format during configuration based on your application's quality and bandwidth requirements.

This table lists the available formats and their use cases.

| Format | Bytes per Pixel | Characteristics | Use Case |
|---|---|---|---|
| `RGB8` | 3 | 8-bit per channel (R, G, B). Compact 24-bit color. | Visualization, JPEG storage, simple processing |
| `RGB12` | 6 | 12-bit per channel (2 bytes each). Full color fidelity. | Scientific analysis, archival, high-quality reconstruction, computer vision |
| `BayerRG12` | 2 | 12-bit Bayer pattern (unpacked). Requires demosaicing. | Full color fidelity in smaller package, with complexity of unpacking 12bits |
| `BayerRG12packed` | 1.5 | 12-bit Bayer (packed, 1.5 bytes per pixel). | Smallest full fidelity package. Lowest bandwidth usage. Requires complex unpacking |
| `BayerRG8` | 1 | 8-bit Bayer pattern. Single-byte per pixel. | Legacy systems, extreme bandwidth constraints at cost of color fidelity. Use if you only intend to convert to Jpg in post processing |

The main difference here is the color channel depth: 8bit vs 12bit. In short, 8bit images have one color channel ranging from black to red in the range [0, 255]. 12 bit images have for the same black to red range more colors: [0, 4095].

Bayer formats contain raw sensor data with one color channel per pixel. To convert Bayer to RGB, you must apply a demosaicing algorithm. The `SaveAsJpeg()` method automatically demosaics Bayer data when saving.

It is most efficient to tune this format to your needs. Changing the format later requires more work and transformations. For example: `factory.SetImageOutputFormat(xavia::sdk::PixelFormat::RGB8);` combined with `SaveAsJpeg()` uses less resources as `factory.SetImageOutputFormat(xavia::sdk::PixelFormat::Ba` combined with `SaveAsJpeg()`, even though the result is the same.

#### 8.2.2.3 How to write as jpeg

The `SaveAsJpeg()` method converts any pixel format to 8-bit JPEG and writes it to disk:

```
image->SaveAsJpeg(
  L"image_" + std::to_wstring(frameNumber) + L".jpg",  // File path (wide string)
  90,    // Quality (0-100, recommended: 80-95)
  false  // Chromatic subsampling
);
```

> **Important**: jpeg's are always converted to RGB8. Storing as jpeg will discard the extra color fidelity of the 12 bit recording. We advise to use jpeg recording only for cases

where the main purpose of the camera images is to visualize the colors for humans. If the goal is to process the data by a computer, then using the full fidelity might be better and we advise to store as RGB12 using the `Serialize()` function. Assuming your algorithm needs this color range.

**Parameters**:

- `filePath`: Wide string path where the JPEG file will be written. Must be an absolute or valid relative path.
- `quality`: JPEG quality 0-100. Values 85-95 provide excellent quality with good compression. Lower values (70-80) reduce file size at slight quality cost.
- `chromaticSubsampling`: Boolean flag. Set to `true` to enable 4:2:0 chroma subsampling. Set to `false` for maximum quality (4:4:4 = no subsampling).

We advise to enable chormatic subsampling. Experiments have shown that enabling subsampling has a big and positive impact on the compression speed. The cost on color fidelity is not visible for the human eye.

**Example: Record frames as JPEG without threading**:

This is the simplest form of jpeg recording:

```
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  const auto& metadata = image->GetMetaData();
  image->SaveAsJpeg(L"image_" + std::to_wstring(metadata.frameNumber) + L"_rgb8.jpg", 90, true);
}

int main()
{
  // setup
  factory.SetImageCallback(imageCallback);
  factory.SetImageOutputFormat(xavia::sdk::PixelFormat::RGB8); // to optimize performance.

  //   ...

  camera->Start();
  std::this_thread::sleep_for(std::chrono::seconds(30));
  camera->Stop();
  return EXIT_SUCCESS;
}
```

On a laptop with I7-13620H processor, SIMD support and SSD storage, this is capable of storing 100% of the frames without blocking the SDK thread too long. If your hardware is less powerful, or you are running additional software, consider moving to the multi threaded example.

**Example: Record frames as JPEG with threading**:

Example where JPEG writing is done in a separate thread:

```
#include <thread>
#include <queue>
#include <mutex>
#include <atomic>

std::queue<std::unique_ptr<xavia::sdk::IImage>> jpegQueue;
std::mutex jpegMutex;
std::atomic<bool> running{true};
```

```cpp
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  // Quick enqueue in callback
  {
    std::lock_guard<std::mutex> lock(jpegMutex);
    jpegQueue.push(std::move(image));
  }
}

void jpegWriterThread()
{
  while (running)
  {
    std::unique_ptr<xavia::sdk::IImage> image;
    {
      std::lock_guard<std::mutex> lock(jpegMutex);
      if (jpegQueue.empty())
      {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        continue;
      }
      image = std::move(jpegQueue.front());
      jpegQueue.pop();
    }

    try
    {
      const auto& meta = image->GetMetaData();
      image->SaveAsJpeg(
        L"frame_" + std::to_wstring(meta.frameNumber) + L".jpg",
        90,   // Quality
        true  // Chromatic subsampling enabled for speed
      );
    }
    catch (const std::exception& e)
    {
      std::wcerr << L"Failed to save JPEG: " << e.what() << std::endl;
    }
  }
}

int main()
{
  // ... setup camera ...

  std::thread jpegWriter(jpegWriterThread);

  camera->Start();
  std::this_thread::sleep_for(std::chrono::seconds(30));
  camera->Stop();

  running = false;
  jpegWriter.join();
```

```
  return EXIT_SUCCESS;
}
```

**Error handling of SaveAsJpeg**:

SaveAsJpeg() throws std::exception on failure. Wrap it in try-catch:

```
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  try
  {
    image->SaveAsJpeg(L"output.jpg", 90, false);
  }
  catch (const std::exception& e)
  {
    std::wcerr << L"Error: " << e.what() << std::endl;
    // Handle disk full, permission denied, path not found, etc.
  }
}
```

#### 8.2.2.4 Performance Callback timing constraints:

Images are delivered via callback on the SDK's internal worker thread. If your callback takes too long to execute, frames may be dropped and the camera may become unresponsive. As a guideline, keep callback execution time under a 50 milliseconds. If this is not possible, then offload the work to a queue with multiple workers or split the work over multiple threads.

Example of offloading to a queue:

```
#include <queue>
#include <mutex>
#include <thread>

std::queue<std::unique_ptr<xavia::sdk::IImage>> imageQueue;
std::mutex queueMutex;

void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  // Callback: quick enqueue operation
  {
    std::lock_guard<std::mutex> lock(queueMutex);
    imageQueue.push(std::move(image));
    // here you should also handle dropping frames if you queue is at the limit.
  }
}

// In worker thread(s):
void processImages()
{
  while (keepWorking)
  {
    std::unique_ptr<xavia::sdk::IImage> image;
    {
      std::lock_guard<std::mutex> lock(queueMutex);
      if (imageQueue.empty()) continue;
      image = std::move(imageQueue.front());
      imageQueue.pop();
```

```
    }

    // Now process heavy work on this thread
    image->SaveAsJpeg(L"output.jpg", 90, false);
  }
}
```

**Frame buffering**:

The SDK maintains an internal buffer of pending images. Configure the buffer size before building the sensor:

```
factory.SetImageBufferSize(5);  // Default: 5 frames
```

A larger buffer accommodates temporary processing delays but uses more memory and will provide stale data. A smaller buffer may drop frames if the callback stalls.

**Multi-threaded JPEG saving**:

On a desktop with I7 range processor, SIMD support and SSD storage, saving to jpeg in the SDK thread appears feasible, assuming the image format was optimized for it. But it is at all times advised to do your own testing. If your hardware is not capable of processing all frames, use one or more dedicated worker threads (see the example in How to write as jpeg). This prevents callback blocking and maintains consistent frame acquisition. The `SaveAsJpeg()` function is thread safe and can be called on different images from different threads in parallel.

**Chromatic subsampling performance**:

Enabling chromatic subsampling (`true`) in `SaveAsJpeg()` speeds up JPEG encoding by approximately 60% while slightly reducing color detail perception. For real-time recording on resource-constrained systems, enable subsampling.

**Network bandwidth**:

Camera streaming requires a gigabit network with jumbo frames enabled. Ensure your network is configured correctly and dedicated to sensor traffic for best performance. Multiple applications accessing the same sensor increase contention; prefer single-process designs with shared pointers if using multiple consumers.

**Maximizing throughput**:

If you encounter issues with storing camera images, then follow these guidelines:

- On storage constrained systems: use RGB8 image format and store as jpg. These have the smallest file size, but use more processing power during the recording for the demosaic and compression steps.
- On processing constrained systems: user BayerRG8, BayerRG12 or BayerRG12packed and store using the `Serialize()` function. No processing is required, at the cost of more storage and post processing to make the images useful. Storage requirements are BayerRG8 $<$ BayerRG12packed $<$ BayerRG12. Respectively 1.6MB $<$ 2.4MB $<$ 3.1MB per image. Speed will then be constrained by the storage device.

**8.2.2.5 Serialization** To preserve the full color depth and avoid compression, it is also possible to store the raw byte buffer of the image. This is done by using the `Serialize()` function on the image. This is useful for offline analysis or archival:

```
void imageCallback(std::unique_ptr<xavia::sdk::IImage> image)
{
  try
  {
    auto serialized = image->Serialize();
```

```
    // Write to file
    std::ofstream file(L"frame_" + std::to_wstring(frameNumber) + L".bin", std::ios::binary);
    file.write(reinterpret_cast<const char*>(serialized.data()), serialized.size());
    file.close();
  }
  catch (const std::exception& e)
  {
    std::wcerr << L"Serialization failed: " << e.what() << std::endl;
  }
}
```

**When to use this format**:

- Preserving full 12-bit color information for scientific analysis
- Creating an archive for later processing without the sensor
- Debugging image acquisition issues by examining raw sensor output
- Post-processing where bandwidth-intensive transmission is not a constraint

**Binary content**:

The `Serialize()` function will return a buffer of bytes (as `std::uint8_t`). The format has a fixed 64-byte header followed by pixel data. The interpretation of these bytes is:

| Offset | Size (bytes) | Type | Field | Description |
|---|---|---|---|---|
| 0 | 1 | uint8_t | Format Version | Always 0x01. Indicates the serialization format version. |
| 1–8 | 8 | uint64_t | Frame Timestamp (us) | Timestamp from camera (microseconds) at start of exposure. |
| 9–16 | 8 | uint64_t | Received Timestamp (us) | Timestamp when frame was received (microseconds). |
| 17–20 | 4 | uint32_t | Frame Number | Sequential frame counter. |
| 21–24 | 4 | uint32_t | Width | Image width in pixels. |
| 25–28 | 4 | uint32_t | Height | Image height in pixels. |
| 29 | 1 | uint8_t | Pixel Format | Enum value (0=Unknown, 1=BayerRG8, 2=BayerRG12, 3=BayerRG12packed, 4=RGB8, 5=RGB12). |
| 30–63 | 34 | Reserved | Padding | Reserved for future use. All bytes are 0x00. |
| 64+ | Variable | uint8_t[] | Pixel Data | Raw pixel data. Size depends on width × height × bytes-per-pixel of the format. |

The `PixelFormat` enum values are: `Unknown=0`, `BayerRG8=1`, `BayerRG12=2`, `BayerRG12packed=3`, `RGB8=4`, `RGB12=5`. Refer to the Image formats section for bytes-per-pixel details.

> Note that the SDK also provides a `Deserialize()` function to read this format. See the Deserialization section.

### 8.2.3 Offline features

Offline features allow processing of previously saved images without a live camera connection.

**8.2.3.1 Deserialization** Load images saved in binary format for offline analysis. To do this, first create an empty image using the static factory `xavia::sdk::CreateImage();`; Then use the `Deserialize()` function of the image to interpret the buffer retrieved from file, database or elsewhere. Once deserialized, access pixel data and metadata just like live images:

```
#include <fstream>
#include <xavia/iImage.h>

std::vector<uint8_t> loadXImage(const std::wstring& filePath)
{
  std::ifstream file(filePath, std::ios::binary | std::ios::ate);
  if (!file.is_open())
    throw std::runtime_error("Cannot open file");

  size_t fileSize = file.tellg();
  file.seekg(0, std::ios::beg);

  std::vector<uint8_t> data(fileSize);
  file.read(reinterpret_cast<char*>(data.data()), fileSize);
  return data;
}

void analyzeOfflineImage(const std::wstring& xImgPath)
{
  try
  {
    auto data = loadXImage(xImgPath);
    auto image = xavia::sdk::CreateImage();
    image->Deserialize(data);

    // Access metadata and pixel data
    const auto& metadata = image->GetMetaData();
    const auto& pixelData = image->GetPixelData();

    std::wcout << L"Frame: " << metadata.frameNumber << L", ";
    std::wcout << metadata.width << L"x" << metadata.height << std::endl;

    // Perform analysis or re-encode as JPEG
    image->SaveAsJpeg(L"analyzed_" + std::to_wstring(metadata.frameNumber) + L".jpg", 90, true);
  }
  catch (const std::exception& e)
  {
    std::wcerr << L"Deserialization failed: " << e.what() << std::endl;
  }
}
```

**Offline processing workflow**:

1. Capture images in binary format during sensor operation
2. Transfer files to processing system
3. Deserialize and analyze each frame
4. Re-encode as needed for distribution or visualization

### 8.2.4 Error handling

Camera operations can fail for various reasons: network issues, hardware problems, timeout, or permission conflicts. The SDK provides two mechanisms for error handling: synchronous exceptions and asynchronous callbacks.

**Synchronous errors** occur during method calls like `Start()` and `Stop()`. These are reported as thrown exceptions and must be caught immediately.

**Asynchronous errors** occur during runtime operation (e.g., frame transmission failure). These are reported via the error callback registered on the factory.

**8.2.4.1 Handling start/stop exceptions** The `Start()` and `Stop()` methods throw `xavia::sdk::Exception` on failure:

```
try
{
  camera->Start();
}
catch (const xavia::sdk::Exception& e)
{
  std::wcerr << L"Failed to start camera: " << e.what() << std::endl;

  // Determine error type for recovery
  switch (e.GetType())
  {
    case xavia::sdk::SensorError::UNABLE_TO_START:
      std::wcerr << L"Camera hardware not responding. Check power and network." << std::endl;
      break;
    case xavia::sdk::SensorError::TIMEOUT:
      std::wcerr << L"Camera startup timed out. Try increasing timeout or checking network." <<
      break;
    default:
      std::wcerr << L"Unknown error: " << e.what() << std::endl;
  }
  return EXIT_FAILURE;
}
```

**8.2.4.2 Handling asynchronous runtime errors** Register an error callback on the factory to handle runtime errors:

```
void errorCallback(const xavia::sdk::Exception& e)
{
  std::wcerr << L"Async error: " << e.what() << std::endl;

  switch (e.GetType())
  {
    case xavia::sdk::SensorError::INVALID_FRAME:
      std::wcerr << L"Failed to retrieve frame from camera hardware" << std::endl;
      break;
    default:
      std::wcerr << L"Unspecified error: " << e.what() << std::endl;
      break;
  }
```

```
}

// In configuration phase
xavia::sdk::SensorFactory factory;
factory.SetErrorCallback(errorCallback);
```

#### 8.2.4.3 Trouble shooting

Here are some errors that you can expect to see.

- **WARNING: Received incomplete image, dropping frame**: This error indicates that some of the camera images were corrupted during network transport or arrived incomplete. See if you can reduce the network or processor load. Use the ArenaView to tweak the network resend options and check the vendor support topics (this on for example).
- **GC_ERR_ACCESS_DENIED: Unable to start acquisition**: Full message:

  ```
    Exception: Exception: An error occurred that is not specifically handled by this SDK. Ext
  -1005 GenTL::GC_ERR_ACCESS_DENIED(-1005) :
    GenTL::DataStreamGevLwf::StartAcquisition():242: GenTL::GC_ERR_ACCESS_DENIED(-1005)
  -1005 GenTL::GC_ERR_ACCESS_DENIED(-1005) :
    GenTL::HALGev::WriteReg():1712: address: 0x960, ACK = 0x83, status = 0x8006, ack_id = 0xb
  ```

  Indicates that the SDK was unable to start the camera. Usually this means that another process is busy with the camera (e.g. ArenaView).

---

### 8.2.5 Single frame acquisition

The SDK delivers frames via continuous callbacks. For applications that need to capture a single frame on-demand (e.g., a "take photo" button), you can implement a helper class that wraps the callback mechanism.

**Helper class for single frame acquisition**:

```
#include <xavia/iImage.h>
#include <memory>
#include <mutex>
#include <condition_variable>
#include <atomic>
#include <chrono>

class SingleFrameCapture
{
public:
  /**
   * Request and wait for the next frame from the camera stream.
   * @param timeoutMs Maximum time to wait for the frame in milliseconds (default: 5000 ms)
   * @return Unique pointer to the captured image
   * @throws runtime_exception("timeout") if not frame is received
   */
  std::unique_ptr<xavia::sdk::IImage> GetSingleFrame(uint32_t timeoutMs = 5000)
  {
    // Signal that we want to capture the next frame
    frameRequested = true;

    // Wait for callback to deliver the frame
    std::unique_lock<std::mutex> lock(mutex);
    bool frameArrived = condition.wait_for(
```

```
      lock,
      std::chrono::milliseconds(timeoutMs),
      [this]() { return capturedFrame != nullptr; }
    );

    if (!frameArrived)
    {
      frameRequested = false;
      throw std::runtime_exception("timeout");
    }

    // Return the captured frame
    return std::move(capturedFrame);
  }

  /**
   * This is the callback to register with the factory.
   * Only processes frames when a single frame is requested.
   */
  void FrameCallback(std::unique_ptr<xavia::sdk::IImage> image)
  {
    // Check if a single frame is requested
    if (!frameRequested)
    {
      // Ignore this frame (discard it)
      return;
    }

    // A frame is requested, store it and notify
    {
      std::lock_guard<std::mutex> lock(mutex);
      capturedFrame = std::move(image);
      frameRequested = false;
    }

    // Wake up the thread waiting in GetSingleFrame()
    condition.notify_one();
  }

private:
  std::mutex mutex;
  std::condition_variable condition;
  std::atomic<bool> frameRequested{false};
  std::unique_ptr<xavia::sdk::IImage> capturedFrame;
};
```

**Usage example**:

```
int main()
{
  auto frameCapture = std::make_unique<SingleFrameCapture>();

  // PART 1: CONFIGURATION
  xsdk::SensorFactory factory;
  factory.SetImageCallback(
```

```
    [&](std::unique_ptr<xsdk::IImage> img) { frameCapture->FrameCallback(std::move(img)); }
  );

  // PART 2: RUNTIME
  auto sensor = factory.Build();
  auto camera = sensor->GetCamera();

  if (!camera)
  {
    std::wcerr << L"Camera not available" << std::endl;
    return EXIT_FAILURE;
  }

  camera->Start();

  try
  {
    // Simulate a "take photo" event
    std::wcout << L"Waiting for first frame..." << std::endl;
    auto frame1 = frameCapture->GetSingleFrame(5000);  // Wait up to 5 seconds
    std::wcout << L"Frame captured: " << frame1->GetMetaData().width
               << L"x" << frame1->GetMetaData().height << std::endl;
    frame1->SaveAsJpeg(L"capture_1.jpg", 90, true);
  }
  catch(...)
  {
    std::wcerr << L"Timeout waiting for frame" << std::endl;
  }

  camera->Stop();
  return EXIT_SUCCESS;
}
```

---

## 8.3 Problems

### 8.3.1 Symbolic links

On linux targets, the library files of the camera SDK rely heavily on symbolic links for SONAME resolution. These symbolic links should be included in the archive. However, in some occasion these can get lost or corrupted (e.g. if the archive was opened on Windows). If you see errors during link time like "cannot find libarena.so" and similar, then the symbolic links are gone or corrupted. To solve this, inspect the lib folder of where you have unpacked the SDK. You will find library files like `libarena.so.1.0.0`. The symbolic links you should create are:

- `libarena.so.1.0 -> libarena.so.1.0.0`
- `libarena.so.1 -> libarena.so.1.0`
- `libarena.so -> libarena.so.1`

To do this manually for all 30+ libraries is a hassle. Instead you can use these techniques:

In a bash shell you can use.

```
#!/usr/bin/env bash

set -euo pipefail
```

```
DIR="${1:-.}"

cd "$DIR"

for fullfile in *.so.*.*.*; do
  [[ -e "$fullfile" ]] || continue

  #extract parts
  base="${fullfile%%.so.*}"
  version="${fullfile##*.so.}"
  major="${version%%.*}"
  minor_patch="${version#*.}"
  minor="${minor_patch%%.*}"

  so_minor="${base}.so.${major}.${minor}"
  so_major="${base}.so.${major}"
  so_plain="${base}.so"

  echo "Processing: $fullfile"

  # create symlinks
  ln -sf "${fullfile}" "${so_minor}"
  ln -sf "${so_minor}" "${so_major}"
  ln -sf "${so_major}" "${so_plain}"
done
```

If the `[[ -e "$fullfile" ]] || continue` gives errors, replace it with:

```
if [ ! -e "$fullfile" ]; then
    continue
fi
```

Alternatively, you can use cmake to do this.

Use the following pattern to create the symbolic links at configure time:

```
file(GLOB versioned_libs "${lib_dir}/*.so*")

message(STATUS "Creating build-time symlinks for libraries in ${lib_dir}:")

foreach(versioned_lib ${versioned_libs})
    # Extract just the filename
    get_filename_component(lib_name "${versioned_lib}" NAME)

    # Skip if it's already a symlink in the source directory
    if(IS_SYMLINK "${versioned_lib}")
        continue()
    endif()

    # Match fully versioned library name (e.g., libarena.so.1.0.0)
    if(lib_name MATCHES "^(.+\\.so)\\.[0-9]+(\\.[0-9]+)*(\\.[0-9]+)$")
        # This is a fully versioned library, create shorter symlinks to it
        set(full_versioned_file "${versioned_lib}")
        set(base_name "${CMAKE_MATCH_1}")

        # Extract all partial versions and create symlinks
```

```
        # For libarena.so.1.0.0, create: libarena.so.1.0 and libarena.so.1 and libarena.so
        set(current_name "${lib_name}")

        while(current_name MATCHES "\\.[0-9]+$")
            # Remove the last version component
            string(REGEX REPLACE "(\\.[0-9]+)$" "" link_name "${current_name}")

            # Create symlink in build directory pointing to the full versioned file
            if(NOT EXISTS "${CMAKE_BINARY_DIR}/${link_name}")
                file(CREATE_LINK "${full_versioned_file}" "${CMAKE_BINARY_DIR}/${link_name}" SYM
            endif()

            set(current_name "${link_name}")
        endwhile()
    endif()
endforeach()
```

You can also ask cmake to create these symbolic link at install time (e.g. if you need to distribute them further.):

```
function (create_symlink_at_install target_path)
    # create a symlink at install time using CMake's install(CODE) command

    # as long as we have versions, remove the last one and create symlink to previous one
    # so so.1.0.0 -> so.1.0 -> so.1 -> so
    while(NOT target_path MATCHES "\\.so$")
        string(REGEX REPLACE "(\\.so.*)(\\.[0-9]+)$" "\\1" link_path "${target_path}")
        install(CODE "execute_process(COMMAND ${CMAKE_COMMAND} -E create_symlink ${target_path}
        set(target_path ${link_path})
    endwhile()
endfunction()
```