Orbbec SDK C ++ API user guide-v1.0

Contents

1 Brief Introduction

1.1 Purpose

This document aims to standardize the usage of API interfaces for the Orbbec SDK 1.8, in order to assist users in integrating it into their projects efficiently and effectively, while preventing issues caused by improper API calls. It focuses on commonly used features.

1.2 terms

Name	Explain
Firmware	Firmware of 3D camera
UnPack	During data packet transmission, such as 12-bit packetization, the SDK unpacks and converts it to 16-bit. The main purpose is to save bandwidth during packet transmission over USB 2.0.
Soft Filter	Software filtering refers to the process of applying software algorithms to process the collected sensor data in order to remove noise and smooth the image.
Disparity	Disparity is to observe the direction difference of the same target from two points with a certain distance.
D2D (Disparity to depth)	Disparity to depth is an image processing technique used to convert disparity information into depth information.
Hardware D2D	Disparity to depth is implemented internally in the device, without occupying the computational power of the host computer.
Software D2D	Disparity to depth, implemented in Orbbec SDK
Depth point cloud	Depth point cloud, the coordinates of points in a three-dimensional world coordinate system, can be transformed into a point cloud using the intrinsic parameters of a Depth camera.
RGBD point cloud	Point cloud with overlaid RGB information
D2C	The translation of "Depth to Color" is a feature that performs per-pixel geometric transformation on a depth image. Its result is aligning the depth image with its corresponding color image through the D2C transformation, allowing us to locate the depth information of a color pixel by using the same image coordinate position of that pixel in the transformed depth image. After the D2C transformation, we generate a depth image of the same size as the target color image, where the image content represents depth data in the coordinate system of the color camera. In other words, it reconstructs a depth image "captured" using the origin and dimensions of the color camera, where each pixel matches the corresponding pixel coordinates of the color camera.
Hardware D2C	Hardware D2C refers to the functionality of performing Depth to Color transformation within the camera itself, with the camera directly outputting the result of the D2C transformation
Software D2C	Performing D2C computation on the host computer side using an SDK.
Frame aggregation (FrameSet)	Combining Depth, IR, and Color frames into a Frameset and invoking it through a pipeline.
LDP	Laser close-range protection

2. Obtain Orbbec SDK data streams

2.1 Create a Context

Description	Context: The Context class manages the Orbbec SDK runtime by allocating and releasing resources, enumerating devices, monitoring device reconnection, and enabling multi-device synchronization.
Header file	#include "libobsensor/ObSensor.hpp"
Implementation	ob::Context ctx;

2.2 Registering Device Disconnection Callback

Description	To set a callback function for device disconnection, use the callback that is triggered when a device is inserted or removed. You can then manage the devices based on the returned lists of added and removed devices.
Header file	#include "libobsensor/ObSensor.hpp"

Implementation	// register device callback
	ctx.setDeviceChangedCallback([](std::shared_ptr<ob::DeviceList> removedList,
	std::shared_ptr<ob::DeviceList> addedList) {
	
	});

2.3 Obtain the Device List

Description	Query the device list and return std::shared_ptr<DeviceList>, pointing to the smart pointer of the device list class.
Header file	#include "libobsensor/ObSensor.hpp"
Implementation	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();

2.4 Open Device

Description	Obtain the smart pointer std::shared_ptr<Device> device that points to the 3D camera Device. There are three methods to obtain the device: Method 1 involves specifying the index number of the enumerated device list. Method 2 involves specifying the SN serial number of the enumerated device list. Method 3 involves specifying the UID of the enumerated device list.
Header file	#include "libobsensor/ObSensor.hpp"

Implementation	// **Method 1:** obtain the device from the device list, such as the device with index 0.
	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDevice(0);
	
	// **Method 2:** obtain the specified device by using the serial number(SN)
	// For example, obtain the device whose SN number is "AE4M73D0040".
	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDeviceBySN("AE4M73D0040");
	
	// **Method 3:** obtain the specified device through the device UID.
	// for example, the device with uid NDSG3958LKHY45.
	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDeviceByUid("NDSG3958LKHY45");

2.5 Obtain Device Information

Description	Obtain the information std::shared_ptr<DeviceInfo> pointing to the device (RGBD) camera. You can query the Pid, Vid, serial number, and other related information of the device.
Header file	#include "libobsensor/ObSensor.hpp"
Implementation	auto deviceInfo = device->getDeviceInfo();

2.6 Create a Pipeline

Description	The Pipeline interface is used for both single-device and multi-device operations. To use it, you must first obtain the specified device from the DeviceList and then bind it to the Pipeline.
Header file	#include "libobsensor/ObSensor.hpp"

Implementation	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDevice(0);
	std::shared_ptr<ob::Pipeline> pipeline(new ob::Pipeline(device));

2.7 Obtain Data Streams

After you use ob::Pipeline(ob_device) to create a Pipeline, you can use it to obtain the specified data streams in Depth, IR, and Color.

2.7.1 StreamProfile

The StreamProfile is the play configuration obtained from the StreamProfileList. It must be set when starting a Pipeline or Sensor stream. Both the Pipeline and Sensor provide methods to retrieve the StreamProfileList.

Pipeline	// source code: include/libobsensor/hpp/Pipeline.hpp std::shared_ptr<StreamProfileList> getStreamProfileList(OBSensorType sensorType);
Sensor	// source code: include/libobsensor/hpp/Sensor.hpp const std::shared_ptr<StreamProfileList> getStreamProfileList();

2.7.2 Obtain the Depth Stream

Description	1. To retrieve all Depth Profiles of a 3D camera, including the resolution, frame rate, and frame format of the streams, use ob::Pipeline. Return the results as std::vector<std::shared_ptr<StreamProfile>>. 2. Obtain the desired Depth Profile for output. 3. Create an ob::Config to configure the Depth stream that the ob::Pipeline should enable. 4. Start the ob::Pipeline with the ob::Config to obtain the Depth stream. 5. Reading the Depth stream: It can be done in either synchronous blocking or asynchronous callback mode. Choose one of the two options.
Header File	#include "libobsensor/ObSensor.hpp"

Implementation	**1. Configure Depth**
	// Enumerate devices and use the first device to create a Pipeline.
	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDevice(0);
	std::shared_ptr<ob::Pipeline> pipe(new ob::Pipeline(device));
	
	// Obtain the list of all Depth Profile of the 3D camera.
	auto profiles = pipe->getStreamProfileList(OB_SENSOR_DEPTH);
	
	// Obtain the Depth Profile you want to output.
	std::shared_ptr<ob::VideoStreamProfile> depthProfile = nullptr;
	depthProfile = profiles->getVideoStreamProfile(640, 400, OB_FORMAT_Y16, 30);
	
	if(depthProfile != nullptr){
	// Create the ob::Config configuration file to configure the ob::Pipeline stream to be enabled;
	std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>();
	config->enableStream(depthProfile);
	}
	
	2.1 Obtain Depth Streams in Synchronous Mode
	// Start the ob::Pipeline to obtain the Depth stream.
	pipe.start(config);
	
	bool readFrame = true;
	while(readFrame) {
	// Wait for up to 100 milliseconds in synchronous blocking mode to obtain the frame set.
	auto frameSet = pipe.waitForFrames(100);
	if(frameSet == nullptr) {
	continuous;
	}

	// Obtain the Depth data frame from the frame set.
	auto depth frame = frameset->depth frame ();
	if(depthFrame){
	// Obtain the width of the Depth frame.
	uint32_t width = depthFrame->width();
	// Obtain the height of the Depth frame.
	uint32_t height = depthFrame->height();
	
	// Obtain the Depth data.
	uint16_t *data = (uint16_t *)depthFrame->data();
	// Obtain the frame data stream size.
	uint32_t dataSize = depthFrame->dataSize();
	// Obtain the hardware timestamp of the Depth frame.
	uint64_t timestamp = depth frame-> timestamp ();
	// Obtain the system timestamp of the Depth frame.
	uint64_t sys timestamp = depth frame-> System timestamp ();
	}
	}
	
	2.2 Obtain Depth sStreams Asynchronously
	
	ob::FrameSetCallback callback = [](std::shared_ptr<ob::FrameSet> frameSet) {
	
	// Obtain the Depth data frame.
	auto depth frame = frameset->depth frame ();
	if(depthFrame) {
	/* Business logic processing */
	}
	};
	
	// Set callback and ob::Config to start the ob::Pipeline to obtain the Depth stream.
	holder->pipeline->start(config, callback);
	
	1. Stop Pipiline
	
	// Stop ob::Pipeline and stop reading Depth streams.
	pipe.stop();

2.7.3 obtain the IR stream

Description	1. Retrieve all IR Profiles of a 3D camera, including the resolution, frame rate, and frame format of the streams, using ob::Pipeline. Return as std::vector<std::shared_ptr<StreamProfile>>. 2. Obtain the desired IR Profile for output. 3. Create an ob::Config configuration file to configure the IR stream that the ob::Pipeline should enable. 4. Start the ob::Pipeline with the ob::Config to obtain the IR stream. 5. Reading the IR stream: It can be done in either synchronous blocking or asynchronous callback mode. Choose one of the two options.
Header File	#include "libobsensor/ObSensor.hpp"

Specific Implementation	**1. Configure IR**
	// Enumerate devices and use the first device to create a Pipeline.
	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDevice(0);
	std::shared_ptr<ob::Pipeline> pipe(new ob::Pipeline(device));
	
	// Obtain the list of all IR Profile of the 3D camera.
	auto profiles = pipe->getStreamProfileList(OB_SENSOR_IR);
	
	// Obtain the IR Profile you want to output.
	std::shared_ptr<ob::VideoStreamProfile> irProfile = nullptr;
	IrProfile = profiles->getVideoStreamProfile(640, 400, OB_FORMAT_Y10, 30);
	
	if(irProfile != nullptr){
	// Create the ob::Config configuration file to configure the ob::Pipeline stream to be enabled;

	std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>();
	config->enableStream(irProfile);
	}
	
	2. Obtain the IR Stream
	2.1 Synchronously Obtain IR Streams
	// Start the ob::Pipeline to obtain the IR stream.
	pipe.start(config);
	
	bool readFrame = true;
	while(readFrame) {
	// Wait for up to 100 milliseconds in synchronous blocking mode to obtain the frame set.
	auto frameSet = pipe.waitForFrames(100);
	if(frameSet == nullptr) {
	continuous;
	}
	
	// Obtain the IR data frame from the frame set.
	auto irFrame = frameSet->irFrame();
	if(irFrame){
	/* Business logic processing */
	// Obtain the width of the IR frame.
	uint32_t width = irFrame->width();
	// Obtain the height of the IR frame.
	uint32_t height = irFrame->height();
	
	// Obtain the IR frame data.
	uint16_t *data = (uint16_t *)irFrame->data();
	// Obtain the frame data stream size.
	uint32_t dataSize = irFrame->dataSize();
	// Obtain the hardware timestamp of the IR frame.
	uint64_t timestamp = irframe-> timestamp ();
	// Obtain the system timestamp of the IR frame.
	uint64_t sys timestamp = irframe-> System timestamp ();
	}
	}
	
	2.2 Asynchronous Acquisition of IR Streams
	ob::FrameSetCallback callback = [](std::shared_ptr<ob::FrameSet> frameSet) {
	
	// Obtain the ir data frame.
	auto irFrame = frameSet->irFrame();
	if(irFrame) {
	/* Business logic processing */
	}
	};
	
	// Set callback and ob::Config to start the ob::Pipeline to obtain the IR stream.
	holder->pipeline->start(config, callback);
	
	3. Stop Pipiline
	// Stop ob::Pipeline and stop IR stream reading.
	pipe.stop();

Notes:

1. To obtain the right IR image instead of the default left IR image for the binocular camera, send a command to switch the IR from left to right. Refer to the parameter configuration section for details.

2. Gemini 2 XL can output left and right IR images simultaneously. For more information, refer to the example code in DoubleInfraredViewer.cpp.

2.7.4 Obtain the Color Stream

Description	1. Retrieve all Color Profiles of a 3D camera, including the resolution, frame rate, and frame format of the streams, using ob::Pipeline. Return as std::vector<std::shared_ptr<StreamProfile>>. 2. Obtain the desired Color Profile for output.

	3. Create an ob::Config configuration file to configure the Color stream that the ob::Pipeline should enable. 4. Start the ob::Pipeline with the ob::Config to obtain the Color stream. 5. Reading the Color stream: It can be done in either synchronous blocking or asynchronous callback mode. Choose one of the two options.
Header File	#include "libobsensor/ObSensor.hpp"

Specific Implementation	**1. Configure Color**
	// Enumerate devices and use the first device to create a Pipeline.
	std::shared_ptr<DeviceList> devices = ctx.queryDeviceList();
	auto device = devices->getDevice(0);
	std::shared_ptr<ob::Pipeline> pipe(new ob::Pipeline(device));
	
	// Obtain the list of all Color Profile of the 3D camera.
	auto profiles = pipe->getStreamProfileList(OB_SENSOR_COLOR);
	
	// Obtain the desired Color stream configuration.
	std::shared_ptr<ob::VideoStreamProfile> colorProfile = nullptr;
	colorProfile = profiles->getVideoStreamProfile(640, 480, OB_FORMAT_RGB, 30);
	
	if(colorProfile != nullptr){
	// Create the ob::Config configuration file to configure the ob::Pipeline stream to be enabled;
	std::shared_ptr<ob::Config> config = std::make_shared<ob::Config>();
	CONFIG->enableStream(colorProfile);
	}
	
	2. Obtain the Color Atream
	2.1 Synchronously Obtain the Color Atream
	// Start the ob::Pipeline to obtain the Color stream.
	pipe.start(config);
	
	bool readFrame = true;
	while(readFrame) {
	// Wait for up to 100 milliseconds in synchronous blocking mode to obtain the frame set.
	Auto frameSet = pipe.waitForFrames(100);
	if(frameSet == nullptr) {
	continuous;
	}
	
	// Obtain the Color data frame from the frame set.
	Auto color frame = frameset->color frame ();
	// If the Color frame is not a null pointer, perform business processing.
	if(colorFrame){
	// Obtain the width of the Color frame.
	uint32_t width = colorFrame->width();
	// Obtain the height of the Color frame.
	uint32_t height = colorFrame->height();
	
	// Obtain the Color data.
	uint8_t *data = (uint8_t *)colorFrame->data();
	// Obtain the frame data stream size.
	uint32_t dataSize = colorFrame->dataSize();
	// Obtain the hardware timestamp of the Color frame.
	uint64_t timestamp = color frame-> timestamp ();
	// Obtain the system timestamp of the Color frame.
	uint64_t sys timestamp = color frame-> System timestamp ();
	}
	}
	
	2.2 Asynchronously Obtain the Color Stream
	
	ob::FrameSetCallback callback = [](std::shared_ptr<ob::FrameSet> frameSet) {
	
	// Obtain the Color data frame.
	Auto color frame = frameset->color frame ();
	// If the color frame is not a null pointer, perform business processing.
	if(colorFrame) {
	/* Business logic processing */
	}
	};

	// Set callback and ob::Config to start the ob::Pipeline to obtain the IR stream.
	holder->pipeline->start(config, callback);
	
	3. Stop Pipiline
	// Stop ob::Pipeline and stop IR stream reading.
	pipe.stop();

2.7.5 Obtain IMU Data

The IMU contains an accelerometer and a gyroscope, and their sampling rates are related. If both sensors are active simultaneously, the output frequency will be determined by the last set sampling rate. Please refer to the [ImuReader Sample] for example code.

Accelerometer unit: m/s ^ 2.
Gyroscope unit: rad/s.

Notes:
1. The purpose of the Imu sample is to obtain the default configuration of the accelerometer and gyroscope. To set the range and frequency, refer to the following code:

2.7.5.1 Gyro

Description	1. Obtain the gyroscope Sensor. 2. Obtain the configuration of the gyroscope you want to output 3. Open Stream. 4. Use the callback function to obtain the gyroscope data. 5. Stop streaming
Header File	#include "libobsensor/ObSensor.hpp"

Specific Implementation	**1. Set the Frame Callback Function**
	ob::FrameCallback gyroFrameCallback = [](std::shared_ptr<ob::Frame> frame) {
	uint64_t timeStamp = frame->timeStamp();
	auto gyroFrame = frame->as<ob::GyroFrame>();
	OBGyroValue value = gyroFrame->value();
	std::cout << "Gyro Frame: {tsp = " << timeStamp
	<<", Temperature =" << gyro frame->temperature()
	<< ", Data [" << value.x << ", "<< value.y << "," << value.z << "]rad/s" << std::endl;
	}
	
	2. Open Stream
	// Obtain the gyroscope Sensor. Note: If the gyroscope Sensor cannot be found, it will be throw exception.
	auto gyroSensor = device->getSensor(OB_SENSOR_GYRO);
	if(gyroSensor) {
	// Query the list of supported streamProfile configurations.
	auto profiles = gyroSensor->getStreamProfileList()
	// Set the range and sampling rate of the gyroscope. For the range and sampling rate supported by each device, please refer to the product manual.
	auto profile = profiles->getGyroStreamProfile(OB_GYRO_FS_1000dps,
	OB_SAMPLE_RATE_200_HZ);
	gyroSensor->start(profile, gyroFrameCallback);
	}
	else {
	std::cout << "Get Gyro Sensor failed ! " << std::endl;
	}
	
	3. Stop Streaming
	// Close the data stream and release resources.
	gyroSensor->stop();
	gyroSensor= nullptr;

2.7.5.2 Accelerator (Accel)

Description	1. Obtain the accelerometer Sensor. 2. Obtain the configuration of the accelerometer to be output. 3. Open stream. 4. Obtain the accelerometer data through the callback function. 5. Stop streaming.
Header File	#include "libobsensor/ObSensor.hpp"

Specific Implementation	**1.Set the Frame Callback Function**
	ob::FrameCallback accelFrameCallback = [](std::shared_ptr<ob::Frame> frame) {
	Car timeStamp = frame->timeStamp();
	auto accelFrame = frame->as<ob::AccelFrame();
	auto value = acceleration Frame->value();
	std::cout << "Accel Frame: {tsp = " << timeStamp
	<<", Temperature ="<<accelframe->temperature()
	<< ", data["<< value.x << ", " << value.y << ", " << value.z
	<< "]m/s^2"
	<< std::endl;
	}
	
	2.Open Stream
	// Obtain the accelerometer Sensor. Note: If the Sensor cannot be found, it will be throw exception.
	auto accelSensor = device->getSensor(OB_SENSOR_ACCEL);
	if(accelSensor) {
	// Query the list of supported playback configurations.
	auto proprofiles = accelSensor->getStreamProfileList();
	// Set the range and sampling rate of the accelerometer. For more information about the range and sampling rate supported by each device, see the product manual.
	auto Profile = profiles->getAccelStreamProfile(OB_ACCEL_FS_2 G,
	OB_SAMPLE_RATE_200_HZ;
	accelSensor->start(profile, accelFrameCallback);
	}
	else {
	std::cout << "Get accel Sensor failed ! " << std::endl;
	}
	
	3.Stop streaming
	// Close the data stream and release resources.
	accelSensor->stop();
	accelSensor = nullptr;

3 Parameter Configuration

3.1 The Steps for Parameter Configuration

3.1.1 Parameter Type

int	Integer type with the suffix INT
float	float type with the suffix FLOAT
bool	Boolean type with the suffix BOOL. Valid values: true and false.
struct	For structure types, with the prefix OB_STRUCT, the parameters can be complex. SDKs often encapsulate the operations for structure type parameters into device methods to reduce complexity during invocation. For example ,To obtain device information (ob_device_info) using the following method:. device->getDeviceInfo()

The device parameters are defined in [include/libobsensor/h/Property.h]. The type is enum and the type name is [C ++ interface] OBPropertyID. Each device parameter corresponds to an enum member.

For example:

```
// Color AE enable switch
OB_PROP_COLOR_AUTO_EXPOSURE_BOOL = 2000,
```

3.1.2 Parameter Permissions

When calling device parameters, it is necessary to determine whether the device supports the specified command operation and whether it supports reading and/or writing with that device parameter. Some device parameters are read-only and cannot be written, while others allow both reading and writing.

```
typedef enum {
    OB_PERMISSION_DENY          = 0,    /**< no permission */
    OB_PERMISSION_READ          = 1,    /**< can read */
    OB_PERMISSION_WRITE         = 2,    /**< can write */
    OB_PERMISSION_READ_WRITE = 3,    /**< can read and write */
    OB_PERMISSION_ANY           = 255, /**< any situation above */
} OBPermission type, ob_permittion_type;
```

For example:

```
if (device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_READ_WRITE)) {
    std::cout<< "Support read and write OB_PROP_COLOR_EXPOSURE_INT"<<std::endl;
} else {
    std::cout<< "Not support read and write OB_PROP_COLOR_EXPOSURE_INT"<<std::endl);
}
```

3.1.3 Maximum, Minimum, Step, Default

For certain integer and floating-point parameters, there are limitations on the maximum value, minimum value, and step size. To set these parameters, you should first obtain their range and step size, verify their validity, and avoid any failures caused by non-compliance.

Concept	Description
Parameter range	Certain int and float parameters have limitations on their maximum and minimum values, as well as their step size. The valid range for these parameters includes both the maximum and minimum values, and the step size must be followed. In contrast, bool parameters can only be true or false, with no restrictions on their maximum or minimum values or step size.
Maximum	The parameter value must be within the specified range, including the maximum value. If the value exceeds the maximum, an error will occur during the setting process.
Minimum	If the set parameter value is less than the minimum value, an error will occur during setting as the reasonable range of the parameter includes the minimum value.
Step	If the parameter has a step size, the new value must be a multiple of the step size within the range of the minimum and maximum values.
Default value	Default parameters value after device startup.

```
typedef struct {
    int32_t cur;     ///< Current value
    int32_t max;     ///< Maximum value
    int32_t min;     ///< Minimum value
    int32_t step;    ///< Step value
    int32_t def;     ///< Default value
} OBIntPropertyRange, ob_int_reperty_range;
```

The definition of OBIntPropertyRange can be found in the header file: include/libobsensor/h/ObTypes.h.

Example:

```
// Query the color exposure value range, default value, and step size.
OBIntPropertyRange range = device->geIntPropertyRange( OB_PROP_COLOR_EXPOSURE_INT);
```

3.1.4 Parameter Setting and Getting

Read device parameters. The permission must be OB_PERMISSION_READ or OB_PERMISSION_READ_WRITE.

An example of getting parameters is as follows:

```
// Obtain the exposure value of the color camera.
    uint32_t value = device->getIntProperty(OB_PROP_COLOR_EXPOSURE_INT);
```

Set device parameters. The permission must be OB_PERMISSION_WRITE or OB_PERMISSION_READ_WRITE.

Examples of parameter settings are as follows:

```
// Set the exposure time to 30000 microseconds.
device->setIntProperty(OB_PROP_COLOR_EXPOSURE_INT, 30000);
```

3.1.5 Comprehensive Example

The comprehensive example provides a method to set the color exposure time. Note that before setting or getting parameters:

1. Whether the set parameters are supported.
2. Whether the parameters have read and write permissions.
3. For more information about the validity of the specified parameters, see [maximum, minimum, step, and default].
4. Set the color exposure time.

```
// 1. Check whether the parameters support writing?
if(device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
// 2. Turn off the color AE. The permission check can be ignored here. If the color exposure value is set, the AE switch must be supported.
    device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, false);
// 3. Set the exposure time of the color camera
    int32_t exposure = 2000;
// 4. Obtain the parameter range and step size
    OBIntPropertyRange range =device->getIntPropertyRange(OB_PROP_COLOR_EXPOSURE_INT);
// 5. Check the validity of parameters
    if(exposure >=range.min && exposure <= range.max && ((exposure -range.min)%range.step == 0))    {
// 6. Set the exposure time.
        device->setIntProperty(OB_PROP_COLOR_EXPOSURE_INT, exposure);
    }
}
```

3.2 Common Parameter Configuration

Section 3.1 outlines the steps for configuring parameters. This section focuses on commonly used parameters, omitting details on parameter support, validity detection, and other complexities. The section provides direct guidance on setting and getting commonly used parameters.

3.2.1 Obtain the Serial Number

To retrieve the device serial number, first acquire the device through the pipeline. Then, obtain the device information from the device and retrieve the serial number from the device information. Similarly, you can obtain other information such as the device name, PID, VID, firmware version, and more from the DeviceInfo.

```
ob::Pipeline pipe;
Auto device = pipe.get device ();
Auto deviceInfo = device->getDeviceInfo();
std::string serialNumber = deviceInfo->serialNumber();
```

3.2.2 Obtain Camera Intrinsic and Extrinsic Parameters

3.2.2.1 Obtain Camera Parameters through the Device

The intrinsic parameters obtained through the Device are the raw intrinsic parameters obtained directly from the device, without depth-to-color (D2C) . It is recommended for non-professionals to obtain parameters from the pipeline.

Use case: When only Depth is enabled and D2C alignment is not required, the intrinsic parameters can be obtained from the Device. The intrinsic parameters obtained from the Device are provided as a list, and the appropriate set of intrinsic parameters should be selected based on the resolution, matching the streaming resolution.

```
ob::Pipeline pipe;
auto device = pipe.get device ();
auto cameraParamList = device->getCalibrationCameraParamList();
// Print the intrinsic parameters.
for(int i = 0; i < cameraParamList->count(); i++) {
OBCameraParam obParam = cameraParamList->getCameraParam (I);
    std::cout << "fx:" << obParam.depthIntrinsic.fx << " fy: " << obParam.depthIntrinsic.fy << " cx:" << obParam.depthIntrinsic.cx
                << " cy: " << obParam.depthIntrinsic.cy
        << " width: " << obParam.depthIntrinsic.width << " height:" << obParam.depthIntrinsic.height << std::endl;

}
```

3.2.2.2 Obtain Camera Parameters through Pipeline

To obtain the camera's intrinsic parameters through the Pipeline, enable the Depth stream and retrieve the intrinsic parameters for Depth. Refer to the DepthViewer sample for obtaining the Depth stream. Additionally, obtain the intrinsic parameters after depth-to-color (D2C) alignment by referring to the

SyncAlignViewer sample, which demonstrates obtaining both the Depth and Color streams.

Note: To obtain camera parameters by Pipiline, you must obtain them after the stream is turned on.

```
ob::Pipeline pipe;
/Please refer to the sample code of DepthViewer or SyncAlignViewer to enable stream
pipe.start(config);
auto camera_param = pipe.getCameraParam();
```

3.2.3 Set Depth Parameters

3.2.3.1 Set Depth AE

```
// Set the Depth for automatic exposure.
if(device->isPropertySupported(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, OB_PERMISSION_READ)) {

    bool isOpen = device->getBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL);
    if(device->isPropertySupported(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, OB_PERMISSION_WRITE)) {
        device->setBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, !isOpen);
    }
    isOpen = device->getIntProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL);
Std::cout << "isOpen:" << isOpen << std::endl;
}
```

3.2.3.2 Set Depth Exposure

Set Depth exposure, prerequisites, you need to turn off the Depth AE. For sample device, setting Depth exposure is equivalent to setting IR exposure.

```
 // Turn off AE
device->setBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_DEPTH_EXPOSURE_INT, OB_PERMISSION_READ)) {
 // Obtain the Depth exposure value.
    int32_t exposure = device->getIntProperty(OB_PROP_DEPTH_EXPOSURE_INT);
    if(device->isPropertySupported(OB_PROP_DEPTH_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
 // Set the Depth exposure value.
        device->setIntProperty(OB_PROP_DEPTH_EXPOSURE_INT, exposure / 2);
    }
}
```

3.2.3.3 Set Depth Gain

To set the exposure for Depth, it is necessary to disable Depth AE (Auto Exposure) as a prerequisite. For sample device, setting Depth gain is equivalent to setting IR gain.

```
 // Turn off AE
    device->setBoolProperty(OB_PROP_DEPTH_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_DEPTH_GAIN_INT, OB_PERMISSION_READ)) {
    // Obtain the Depth gain value.
    int32_t gain = device->getIntProperty(OB_PROP_DEPTH_GAIN_INT);
    if(device->isPropertySupported(OB_PROP_DEPTH_GAIN_INT, OB_PERMISSION_WRITE)) {
        // Set the Depth gain value.
        device->setIntProperty(OB_PROP_DEPTH_GAIN_INT, gain + 1000);
    }
}
```

3.2.3.4 Set the Depth Mirror

```
// Determine whether the API is supported?
if(device->isPropertySupported(OB_PROP_DEPTH_MIRROR_BOOL, OB_PERMISSION_READ)) {
    // Obtain the current mirror.
    bool isOpen = device->getBoolProperty(OB_PROP_DEPTH_MIRROR_BOOL);
    if(device->isPropertySupported(OB_PROP_DEPTH_MIRROR_BOOL, OB_PERMISSION_WRITE)) {
        device->setBoolProperty(OB_PROP_DEPTH_MIRROR_BOOL, !isOpen);
    }
}
```

3.2.3.5 Set the Depth to Flip Up and Down

```
if(device->isPropertySupported(OB_PROP_DEPTH_FLIP_BOOL, OB_PERMISSION_WRITE)) {
        // true: depth is flipped up and down; false: Does not flip
    device->setBoolProperty(OB_PROP_DEPTH_FLIP_BOOL,true);
}
```

3.2.3.6 Set Depth Rotation

The Gemini 2 series of devices support depth rotation and can be rotated at 0, 90, 180, and 270 degrees. The default rotation is set to 0 degrees.

```
// Rotation parameters
Typedef enum {
    OB_ROTATE_DEGREE_0      = 0,        ///< Rotate 0
    OB_ROTATE_DEGREE_90     = 90,       ///< Rotate 90
    OB_ROTATE_DEGREE_180 = 180,     ///< Rotate 180
    OB_ROTATE_DEGREE_270 = 270,     ///< Rotate 270
} ob_rotate_degree_type, OBRotateDegreeType;

if(device->isPropertySupported(OB_PROP_DEPTH_ROTATE_INT, OB_PERMISSION_WRITE)) {
    // Set the rotation angle
    device->setIntProperty(OB_PROP_DEPTH_ROTATE_INT, OB_ROTATE_DEGREE_90);
}
```

3.2.3.7 Disparity to Depth (D2D)

1. Hardware D2D

Hardware D2D is the internal implementation of Disparity to Depth conversion within the device itself. This feature is supported by devices such as Gemini 2, Gemini 2 L, Astra 2, Gemini 2 XL. Before invoking the function, it is necessary to check if the interface property is supported.

```
if(device->isPropertySupported(OB_PROP_DISPARITY_TO_DEPTH_BOOL, OB_PERMISSION_WRITE)) {
    // Parameter: true to enable hardware D2D,false to disable hardware D2D
    device->setBoolProperty(OB_PROP_DISPARITY_TO_DEPTH_BOOL, true);
}
```

2. Software D2D

Software D2D is the internal implementation of Disparity to Depth conversion within the SDK. Software D2D is supported by all sensor device except for TOF device such as Femto Mega and Femto Bolt. Before invoking the function, it is necessary to check if the interface property is supported.

```
if(device->isPropertySupported(OB_PROP_SDK_DISPARITY_TO_DEPTH_BOOL, OB_PERMISSION_WRITE)) {
// Parameter: true to enable software D2D,false to disable software D2D
    device->setBoolProperty(OB_PROP_SDK_DISPARITY_TO_DEPTH_BOOL, true);
}
```

3.2.3.8 Set the Minimum and Maximum Values of Depth

To set the minimum and maximum values for Depth, any Depth values that fall outside this range are assigned a value of 0. This can be used to filter out noise.

1. Set the Minimum Depth

```
if(device->isPropertySupported(OB_PROP_MIN_DEPTH_INT, OB_PERMISSION_WRITE)) {
// Set the minimum Depth in mm.
    device->setIntProperty(OB_PROP_MIN_DEPTH_INT, 100);
}
```

2. Set the Maximum Depth

```
if(device->isPropertySupported(OB_PROP_MAX_DEPTH_INT, OB_PERMISSION_WRITE)) {
// Set the maximum Depth in mm.
    device->setIntProperty(OB_PROP_MAX_DEPTH_INT, 10000);
}
```

3.2.3.9 Set the Unit of Depth (Precision)

The precision of depth measurements is determined by the unit of pixel values in the depth data frame. By adjusting the depth unit, the accuracy of depth measurements can be modified. For instance, if the unit is set to 0.2mm, an object at a distance of 1000mm will have a pixel value of 5000 in the output depth data frame (as 5000 multiplied by 0.2mm equals 1000mm). Similarly, if the unit is set to 0.1mm, the pixel value will be 10000.

```
Typedef enum {
    OB_PRECISION_1MM,     /**< 1mm */
    OB_PRECISION_0MM8,    /**< 0.8mm */
    OB_PRECISION_0MM4,    /**< 0.4mm */
    OB_PRECISION_0MM1,    /**< 0.1mm */
    OB_PRECISION_0MM2,    /**< 0.2mm */
    OB_PRECISION_0MM5,    /**< 0.5mm */
    OB_PRECISION_0MM05, /**< 0.05mm */
    OB_PRECISION_UNKNOWN,
    OB_PRECISION_COUNT,
} OBDepthPrecisionLevel,
```

Gemini2, Gemini2 L, Astra2, Gemini2 XL, and Dabai DCL support depth precision settings.

```
if(device->isPropertySupported(OB_PROP_DEPTH_PRECISION_LEVEL_INT, OB_PERMISSION_WRITE)) {
    //Set the Depth unit to 0.1mm.
    device->setIntProperty(OB_PROP_DEPTH_PRECISION_LEVEL_INT, OB_PRECISION_0MM1);
}
```

Note:

1. After OrbbecSDK 1.7.5, the default output Depth unit is 1mm.
2. Depth data, with unit information, can be obtained from the Depth frame interface, the interface is as follows:

```
Auto frameSet = pipe.waitForFrames(100);
if(frameSet == nullptr) {
    continuous;
}

// Obtain the Depth frame data.
Auto depth frame = frameset->depth frame ();
// Obtain the unit of Depth.
float depthUnit = depthFrame->getValueScale();
```

3.2.3.10 Camera Depth Working Mode

"Depth mode" is short for "camera depth working mode". This mode allows a module to adjust the depth camera for different application scenarios by changing parameters. These scenarios may include high-precision, long-distance, and low-power scenes. Each depth application scenario is independent, and switching between modes does not require a device reboot.

The default depth mode is the current working mode after device startup or reboot. To ensure the correct depth mode is used, it is necessary to switch to the desired mode every time the device starts. Upon initial connection to the device, it is important to verify that the current depth mode matches the target depth mode.

Please note that Gemini2, Gemini2 L, Astra2, Dabai DCL, and Gemini2 XL cameras have different depth modes. Each Depth mode may have varying power consumption, supported resolutions, frame rates, and effective depth detection ranges. Please refer to the product specifications for specific details.

For example, to switch the Gemini 2 device to the [Binned Sparse Default] mode, use the following method:

```
device->switchDepthWorkMode("Binned Sparse Default");
```

Note:

1. For more information about switching the depth working mode, see:(DepthWorkMode sample)

3.2.3.11 Software Filtering

Software filtering is used to remove noise from Depth data.

Note: Filtering parameters should only be set when the software filtering switch is turned on. The SDK already has default filtering parameters based on the resolution, so general users do not need to set them. Simply enable software filtering.

1. Software Filter Switch

```
if(device->isPropertySupported(OB_PROP_DEPTH_SOFT_FILTER_BOOL, OB_PERMISSION_WRITE)) {
// Parameter: true enable software filtering, false disable software filtering
    device->setBoolProperty(OB_PROP_DEPTH_SOFT_FILTER_BOOL, true);
}
```

2. Software Filter Parameter Ssettings

To adjust the software filtering, set the 'maxdiff' parameter. This function does not need to be called by default, unless specific adjustments are required. Technical personnel should provide the appropriate parameter settings.

```
if(device->isPropertySupported(OB_PROP_DEPTH_MAX_DIFF_INT, OB_PERMISSION_WRITE)) {
    device->setIntProperty(OB_PROP_DEPTH_MAX_DIFF_INT, 16);
}
```

Set the "maxSpeckleSize" parameter for software filtering. By default, there is no need to call this function unless there are specific requirements for adjustment. The appropriate parameter settings should be provided by technical personnel.

```
if(device->isPropertySupported(OB_PROP_DEPTH_MAX_SPECKLE_SIZE_INT, OB_PERMISSION_WRITE)) {
    device->setIntProperty(OB_PROP_DEPTH_MAX_SPECKLE_SIZE_INT, 480);
}
```

3.2.4 set IR parameters

3.2.4.1 Set IR AE

```
if(device->isPropertySupported(OB_PROP_IR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_READ)) {
                bool isOpen = device->getBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL);
                if(device->isPropertySupported(OB_PROP_IR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_WRITE)) {
                // true: enable AE, false: disable AE
                device->setBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL, !isOpen);
    }
}
```

3.2.4.2 Set IR Exposure

Set IR exposure, prerequisites, you need to turn off the IR AE.

```
    // Turn off AE first.
device->setBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL, false);
    // Obtain and set the exposure value.
if(device->isPropertySupported(OB_PROP_IR_EXPOSURE_INT, OB_PERMISSION_READ)) {
        // Obtain the IR exposure value.
    int32_t exposure = device->getIntProperty(OB_PROP_IR_EXPOSURE_INT);
    if(device->isPropertySupported(OB_PROP_IR_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
        // Set the IR exposure value.
        device->setIntProperty(OB_PROP_IR_EXPOSURE_INT, exposure / 2);
    }
}
```

3.2.4.3 Set IR Gain

Set the IR gain and prerequisites, and turn off the IR AE.

```
// Turn off AE first.
device->setBoolProperty(OB_PROP_IR_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_IR_GAIN_INT, OB_PERMISSION_READ)) {
    // Obtain the IR gain value.
    int32_t gain = device->getIntProperty(OB_PROP_IR_GAIN_INT);
    if(device->isPropertySupported(OB_PROP_IR_GAIN_INT, OB_PERMISSION_WRITE)) {
        // Set the IR gain value.
        device->setIntProperty(OB_PROP_IR_GAIN_INT, gain+1000);
    }
}
```

3.2.4.4 Set the IR Mirror

```
if(device->isPropertySupported(OB_PROP_IR_MIRROR_BOOL, OB_PERMISSION_READ)) {
    // Obtain an IR mirror.
    bool isMirror = device->getIntProperty(OB_PROP_IR_MIRROR_BOOL);
    if(device->isPropertySupported(OB_PROP_IR_MIRROR_BOOL, OB_PERMISSION_WRITE)) {
```

```
        // IR mirror settings
        device->setIntProperty(OB_PROP_IR_MIRROR_BOOL, !isMirror);
    }
}
```

3.2.4.5 Set IR to Flip Up and Down

```
if(device->isPropertySupported(OB_PROP_IR_FLIP_BOOL, OB_PERMISSION_WRITE)) {
// true: flip up and down; false: do not flip
    device->setBoolProperty(OB_PROP_IR_FLIP_BOOL,true);
}
```

3.2.4.6 Set IR Rotation

Gemini 2 Series devices support Depth rotation, which supports 0-degree, 90-degree, 180-degree, and 270-degree rotation.

```
// Rotation parameters
Typedef enum {
    OB_ROTATE_DEGREE_0      = 0,        ///< Rotate 0
    OB_ROTATE_DEGREE_90     = 90,       ///< Rotate 90
    OB_ROTATE_DEGREE_180 = 180,     ///< Rotate 180
    OB_ROTATE_DEGREE_270 = 270,     ///< Rotate 270
} ob_rotate_degree_type, OBRotateDegreeType;

if(device->isPropertySupported(OB_PROP_IR_ROTATE_INT, OB_PERMISSION_WRITE)) {
      // Set the rotation angle parameter
    device->setIntProperty(OB_PROP_IR_ROTATE_INT, OB_ROTATE_DEGREE_90);
}
```

3.2.4.7 Set IR Left and Right Switching

```
//To determine whether switching the left and right IR channels is supported?
if(pipe.getDevice()->isPropertySupported(OB_PROP_IR_CHANNEL_DATA_SOURCE_INT, OB_PERMISSION_READ_WRITE)) {
// Gemini 2 and Gemini 2 L products support SENSOR_IR to select sensor output. 0 is left IR and 1 is right IR.
    int32_t dataSource = 0;
    pipe.getDevice()->setIntProperty(OB_PROP_IR_CHANNEL_DATA_SOURCE_INT, dataSource);
}
```

3.2.5 Set Color Parameters

The following parameters can be adjusted for a color camera: Auto AE, exposure, gain, auto white balance, color temperature, brightness, sharpness, gamma, saturation, hue, power line frequency, and other parameters. In most cases, default color camera parameters are sufficient to meet the requirements.

3.2.5.1 Set Color AE

```
// Set AE
if(device->isPropertySupported(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_READ)) {
    if(device->isPropertySupported(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, OB_PERMISSION_WRITE)) {
// true: indicates that AE is enabled. false: indicates that AE is disabled.
        device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, true);
    }
}
```

3.2.5.2 Set Color Exposure

Set Color exposure, prerequisites, you need to turn off the Color AE.

```
// Disable AE
device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_READ)) {
      // Obtain the exposure value of the Color camera.
    int32_t exposure = device->getIntProperty(OB_PROP_COLOR_EXPOSURE_INT);
    if(device->isPropertySupported(OB_PROP_COLOR_EXPOSURE_INT, OB_PERMISSION_WRITE)) {
        // Set the exposure value of the Color camera.
        device->setIntProperty(OB_PROP_COLOR_EXPOSURE_INT, exposure / 2);
    }
```

```
}
```

3.2.5.3 Set Color Gain

Set the Color gain and prerequisites. You need to turn off the Color AE.

```
// Disable AE
device->setBoolProperty(OB_PROP_COLOR_AUTO_EXPOSURE_BOOL, false);
if(device->isPropertySupported(OB_PROP_COLOR_GAIN_INT, OB_PERMISSION_READ)) {
        // Obtain the Color camera benefit value.
    int32_t gain = device->getIntProperty(OB_PROP_COLOR_GAIN_INT);
    if(device->isPropertySupported(OB_PROP_COLOR_GAIN_INT, OB_PERMISSION_WRITE)) {
        // Set the Color value of the camera.
        device->setIntProperty(OB_PROP_COLOR_GAIN_INT, gain + 10);
    }
}
```

3.2.5.4 Set the Color Mirror

```
// Set the Color mirror.
if(device->isPropertySupported(OB_PROP_COLOR_MIRROR_BOOL, OB_PERMISSION_READ)) {
    if(device->isPropertySupported(OB_PROP_COLOR_MIRROR_BOOL, OB_PERMISSION_WRITE)) {
        // Parameter: true mirror, false non-mirror
        device->setBoolProperty(OB_PROP_COLOR_MIRROR_BOOL, true);
    }
}
```

3.2.5.5 Set the Color to Flip Up and Down

```
if(device->isPropertySupported(OB_PROP_COLOR_FLIP_BOOL, OB_PERMISSION_WRITE)) {
// true: indicates that the data is flipped up and down. false: indicates that the data is not flipped.
    device->setBoolProperty(OB_PROP_COLOR_FLIP_BOOL, true);
}
```

3.2.5.6 Set Color Rotation

```
// Rotation parameters
Typedef enum {
    OB_ROTATE_DEGREE_0      = 0,        ///< Rotate 0
    OB_ROTATE_DEGREE_90     = 90,       ///< Rotate 90
    OB_ROTATE_DEGREE_180 = 180,     ///< Rotate 180
    OB_ROTATE_DEGREE_270 = 270,     ///< Rotate 270
} ob_rotate_degree_type, OBRotateDegreeType;

if(device->isPropertySupported(OB_PROP_COLOR_ROTATE_INT, OB_PERMISSION_WRITE)) {
        // Set the rotation angle.
    device->setIntProperty(OB_PROP_COLOR_ROTATE_INT, OB_ROTATE_DEGREE_90);
}
```

3.2.5.7 Set Color Automatic White Balance (AWB)

```
if(device->isPropertySupported(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, OB_PERMISSION_READ)) {

    bool isOpen = device->getBoolProperty(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL);
    if(device->isPropertySupported(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, OB_PERMISSION_WRITE)) {
        // true: enable AWB; false: disable AWB
        device->setBoolProperty(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, !isOpen);
}
```

3.2.5.8 Set Color White Balance Parameters

To set the white balance parameter, you must turn off automatic white balance first.

```
// Turn off white balance
device->setBoolProperty(OB_PROP_COLOR_AUTO_WHITE_BALANCE_BOOL, false);
if(device->isPropertySupported(OB_PROP_COLOR_WHITE_BALANCE_INT, OB_PERMISSION_READ)) {
```

```
    int32_t colorWhiteBalance = device->getIntProperty(OB_PROP_COLOR_WHITE_BALANCE_INT);
    if(device->isPropertySupported(OB_PROP_COLOR_WHITE_BALANCE_INT, OB_PERMISSION_WRITE)) {
        // Set the Color white balance parameter. The parameter must meet the range and step limit. Otherwise, an exception is thrown.
    device->setIntProperty(OB_PROP_DEPTH_EXPOSURE_INT, colorWhiteBalance +1 );

}
```

3.2.5.9 Set the Color Brightness

```
if(device->isPropertySupported(OB_PROP_COLOR_BRIGHTNESS_INT, OB_PERMISSION_WRITE)) {
    OBIntPropertyRange range = device->getIntPropertyRange(OB_PROP_COLOR_BRIGHTNESS_INT);
    int32_t brightness = 50
       // Verify parameters.
    if(brightness >= range.min && brightness <=range.max && (brightness - range.min) % range.step == 0) {
          device->setIntProperty(OB_PROP_COLOR_BRIGHTNESS_INT, brightness);
    }

}
```

sharpness (OB_PROP_COLOR_SHARPNESS_INT)、
Gamma(OB_PROP_COLOR_GAMMA_INT)、
saturation (OB_PROP_COLOR_SATURATION_INT)、
hue (OB_PROP_COLOR_HUE_INT)、
auto exposure priority(OB_PROP_COLOR_AUTO_EXPOSURE_PRIORITY_INT)、
backlight compensation(OB_PROP_COLOR_BACKLIGHT_COMPENSATION_INT)、
contrast(OB_PROP_COLOR_CONTRAST_INT) .

This params are similar to the method for adjusting brightness and will not be specifically discussed here。

3.2.5.10 set Color power line frequency

It is necessary to set 50Hz or 60Hz according to the power line frequency of different countries and regions. The purpose of setting the power supply frequency is to prevent Color images from flickering.

```
typedef enum {
    OB_POWER_LINE_FREQ_MODE_CLOSE = 0,    ///< Close
    OB_POWER_LINE_FREQ_MODE_50HZ    = 1,    ///< 50Hz
    OB_POWER_LINE_FREQ_MODE_60HZ    = 2,    ///< 60Hz
} ob_power_line_freq_mode,
    OBPowerLineFreqMode;

    if(device->isPropertySupported(OB_PROP_COLOR_POWER_LINE_FREQUENCY_INT, OB_PERMISSION_WRITE)) {
                int powerFreq = OB_POWER_LINE_FREQ_MODE_50HZ;
                device->setIntProperty(OB_PROP_COLOR_BRIGHTNESS_INT, powerFreq);
        }
```

3.2.6 Device Reboot

```
device->reboot();
```

3.2.7 laser Switch

The laser switch is used to turn on or off the laser.

```
if(device->isPropertySupported(OB_PROP_LASER_BOOL, OB_PERMISSION_WRITE)) {
        // true: indicates that the laser is turned on; false: indicates that the laser is turned off.
      device->setBoolProperty(OB_PROP_LASER_BOOL, true);
  }
```

3.2.8 LDP Switch

The LDP (Laser Distance Protection) module is capable of detecting objects in close proximity. When LDP is enabled, the system detects obstacles within the specified distance

range of the camera and gradually reduces the laser power level until the laser is switched off, ensuring laser protection. This functionality is disabled when LDP is disabled.

```
if(device->isPropertySupported(OB_PROP_LDP_BOOL, OB_PERMISSION_WRITE)) {
        // true: indicates that LDP is enabled. false: indicates that LDP is disabled.
        device->setBoolProperty(OB_PROP_LDP_BOOL, true);
}
```

3.2.9 Obtain LDP Measurement Value

To obtain the LDP (Laser Distance Protection) measurement values when the LDP switch is enabled.

Note: This interface is only effective when called after enabling the laser and opening the Depth or IR stream.

```
if(device->isPropertySupported(OB_PROP_LDP_MEASURE_DISTANCE_INT, OB_PERMISSION_READ)) {
        // The return value indicates the LDP measurement value, in mm.
        int32_t value = device->getIntProperty(OB_PROP_LDP_MEASURE_DISTANCE_INT);
}
```

3.2.10 Obtain LDP Protection Status

To obtain the LDP (Laser Distance Protection) status when the LDP switch is enabled.

Note: This interface is only effective when called after enabling the laser and opening the Depth or IR stream.

```
if(device->isPropertySupported(OB_PROP_LDP_STATUS_BOOL, OB_PERMISSION_READ)) {
        // true: indicates that the LDP is in the protected state. false: indicates that the LDP is in the normal unprotected state.
        bool status = device->getBoolProperty(OB_PROP_LDP_STATUS_BOOL);
}
```

3.2.11 Flood Light Switch

Note: The support for the flood lights is required by the device.

```
if(device->isPropertySupported(OB_PROP_FLOOD_BOOL, OB_PERMISSION_WRITE)) {
        // true: indicates that the ir flood is turned on. false: indicates that the ir flood is turned off.
        bool status = device->setBoolProperty(OB_PROP_FLOOD_BOOL, true);
}
```

3.2.12 Device Time Synchronization

// 1. To synchronize multiple devices, the parameter "repeatInterval" represents the synchronization interval in milliseconds (ms). For example, to synchronize every half hour, the repeatInterval would be set to 30 * 60 * 1000 (ms).

```
context->enableDeviceClockSync(uint64_t repeatInterval);
```

//2. To synchronize a single device, you would need to periodically invoke a time synchronization function, such as calling it once every 60 minutes.

```
device->timerSyncWithHost();
```

3.2.12 Network Device IP Address Modification Settings

```
// To modify the network IP configuration of a device
OBNetIpConfig config;
device->setStructuredData(OB_STRUCT_DEVICE_IP_ADDR_CONFIG, &config, sizeof(OBNetIpConfig));
/**
 * @brief Network device IP address configuration（IPv4）
 */
typedef struct {
    /**
     * @brief DHCP status
     *
     * @note 0: static IP; 1: DHCP
     */
    uint16_t dhcp;

    /**
     * @brief IP address（IPv4，Big-endian：192.168.1.10，address[0] = 192, address[1] = 168, address[2] = 1, address[3] = 10）
```

```
     */
    uint8_t address[4];

    /**
     * @brief Subnet mask (big-endian)
     */
    uint8_t mask[4];

    /**
     * @brief Gateway (big-endian)
     */
    uint8_t gateway[4];
} OBNetIpConfig, ob_net_ip_config, DEVICE_IP_ADDR_CONFIG;
```

4 D2C

Depth to Color (D2C) is a feature that performs a pixel-by-pixel geometric transformation on depth images. The result is the spatial alignment of depth images with their corresponding Color images through the D2C transformation. This allows us to locate the depth information corresponding to any pixel in the Color image by finding the same image coordinate position in the D2C-transformed depth image. After applying D2C, a depth image of the same size as the target Color image is generated, where the image content represents depth data in the Color camera coordinate system, where each pixel corresponds to the corresponding pixel coordinate of the Color camera.

D2C can be implemented in two ways: hardware and software. Hardware D2C is implemented internally within the device and does not require any additional computing resources from the host computer. Software D2C is implemented on the host computer side as an SDK software library and requires additional computing resources from the host computer. In cases where the host computer has limited computing resources, it is recommended to use hardware D2C. If the host computer has sufficient computing resources and there is a higher requirement for D2C accuracy, it is recommended to use software D2C.

To use the D2C functionality, both Color and depth data streams must be enabled. You can refer to the "SyncAlignViewer" example in the Orbbec SDK, located in the Examples directory.

```
Typedef enum {
    ALIGN_DISABLE,          // disable D2C
    ALIGN_D2C_HW_MODE, // Hardware D2C
    ALIGN_D2C_SW_MODE, // Software D2C
} OBAlignMode, ob_align_mode;
```

```
//For Example: Set hardware D2C
config->setAlignMode(ALIGN_D2C_HW_MODE);
 pipe.start(config);
```

Note: How to determine the selected Depth resolution and Color resolution support hardware D2C alignment?

The code is as follows:

```
ob::Pipeline pipe;
auto                profiles = pipe.getStreamProfileList(OB_SENSOR_COLOR);
// Use 1920x 1080 resolution as an example
auto colorProfile = profiles->getVideoStreamProfile(1920, 1080, OB_FORMAT_MJPEG, 30);
// Obtain the list of colorProfile supported by the depthProfile. Hardware D2C:ALIGN_D2C_HW_MODE
auto depthProfileList = pipe.getD2CDepthProfileList(colorProfile, ALIGN_D2C_HW_MODE);
auto count                      = depthProfileList->count();
for(size_t i = 0; i < count; i++) {
    auto depthProfile = depthProfileList->getProfile(i)->as<ob::VideoStreamProfile>();

    int width    = depthProfile->width();
    int height = depthProfile->height();
    int format = depthProfile->format();
    int fps       = depthProfile->fps();

     // Print the Depth data stream information that supports D2C, including the width and height of the image.
    std::cout << "width：  " << width << " height: " << height << " format: " << format << " fps:" << fps << std::endl;
    }
```

Similarly, how do I know that the selected depth resolution and Color resolution support software D2C alignment? Simply change the code from ALIGN_D2C_HW_MODE to ALIGN_D2C_SW_MODE.

5 Point Cloud

The point cloud reference C++ sample is called "PointCloud". It contains two types of point cloud: a depth point cloud containing only 3D information, and an RGBD point cloud containing both 3D and RGB texture information.

5.1 Depth Point Cloud

1. To enable the Depth stream, refer to the sample code for obtaining the Depth stream and enable the Depth stream with the corresponding resolution.
2. Create a point cloud object

```
ob::PointCloudFilter pointCloud;
```

3. Obtain camera intrinsic parameters

```
auto cameraParam = pipeline.getCameraParam();
pointCloud.setCameraParam(cameraParam);
```

4. Obtain point cloud data

```
// Wait for frames in synchronous blocking mode, with a maximum wait of 100ms
auto frameset = pipeline.waitForFrames(100);
// Obtain Depth unit.
auto depthValueScale = frameset->depthFrame()->getValueScale();
// Set the Depth unit to the point cloud Filter.
pointCloud.setPositionDataScaled(depthValueScale);
// Set the type of the point cloud to the output Depth point cloud.
pointCloud.setCreatePointFormat(OB_FORMAT_POINT);
// To generate point cloud data using a point cloud filter.
std::shared_ptr<ob::Frame> frame = pointCloud.process(frameset);
// Save the point cloud data as .ply file.
savePointsToPly(frame, "DepthPoints.ply");
```

5.2 RGBD Point Cloud

1. Enable Depth and Color data streams and enable D2C alignment (both hardware and software). For more information, see D2C.
2. Create a point cloud object

```
ob::PointCloudFilter pointCloud;
```

3. Obtain camera intrinsic parameters

```
auto cameraParam = pipeline.getCameraParam();
pointCloud.setCameraParam(cameraParam);
```

4. Obtain point cloud data

```
// Wait for a frame in synchronous blocking mode for up to 100ms.
Auto frameset = pipeline.waitForFrames(100);
// Obtain the unit of Depth.
auto depthValueScale = frameset->depthFrame()->getValueScale();
// Set the Depth unit to the point cloud Filter.
pointCloud.setPositionDataScaled(depthValueScale);
// Set the type of the point cloud to output the RGBD point cloud.
pointCloud.setCreatePointFormat(OB_FORMAT_RGB_POINT);
// Call the point cloud Filter to generate point cloud data.
std::shared_ptr<ob::Frame> frame = pointCloud.process(frameset);
// Save the point cloud data as a. ply file.
Saver GB points Toby (frame, "RGB points.ply");
```

6 Video Recording and Playback

Orbbec SDK the file format of the recording is rosbag, the Depth recording will be lossless compressed inside the SDK. The files recorded by the Orbbec SDK can only be played back using the playback interface provided by the Orbbec SDK. They are not compatible with standard ROS playback methods.

6.1 Video Recording

The Orbbec SDK supports the recording of video frame data. If D2C (Depth to Color) is enabled during recording, the recorded files will contain D2C aligned data. For an example of recording, please refer to the "Recording Sample" provided with the SDK. The following example explains the recording functionality using the recorded D2C depth and Color images from the Femto Mega camera. The recording functionality for other products is similar to Femto Mega and you can refer to the product manual to change the

resolution.

Note: If you want to record data after D2C, make sure that the resolution of the enabled Depth and Color streams supports alignment. For more information, see D2C.

```
ob::Pipeline pipe;

// 1. Depth configuration
std::shared_ptr<ob::Config> config               = std::make_shared<ob::Config>();
auto      depthProfiles          = pipe.getStreamProfileList(OB_SENSOR_DEPTH);
auto depthProfile    = depthProfiles->getVideoStreamProfile(640, 576, OB_FORMAT_Y16, 30);
// 2. Color configuration
auto colorProfiles = pipe.getStreamProfileList(OB_SENSOR_COLOR);
auto colorProfile    = colorProfiles->getVideoStreamProfile(1920, 1080, OB_FORMAT_RGB, 30);
// 3. Enable Depth stream
config->enableStream(depthProfile);
// 4. Enable Color stream
config->enableStream(colorProfile);

// 5. Enable D2C
config->setAlignMode(ALIGN_D2C_HW_MODE);
// 6. Open stream
pipe.start(config);
// 7. Enable video recording
pipe.startRecord("./OrbbecPipeline.bag");

while(1)
{
// Wait for the video to complete. For more information, see Record Sample.
}

// 8. Stop recording
pipe.stopRecord();
// 9. Stop streaming
pipe.stop();
```

6.2 Video Playback

For video playback, please refer to the "Playback Sample" provided with the Orbbec SDK. This sample also supports playback of video files recorded using the OrbbecViewer tool. When playing back files recorded with OrbbecViewer, be sure to rename the recorded file to match the filename expected by the Playback Sample.

7 Log management

Each log management interface has a log level parameter, and the log level is determined by the last called interface.

To set the global log level, you can use the setLoggerSeverity function, which simultaneously controls the log level for console output, file output and the log callback function.

The following three log output methods are independent and setting one will not affect the others.

1、 Set the log output to the console (setLoggerToConsole function).
2、 Set output to file (setLoggerToConsole function)
3、 Set log output from callback function (setLoggerToCallback function)

7.1 set the log level

```
typedef enum {
    OB_LOG_SEVERITY_DEBUG, /**< debug */
    OB_LOG_SEVERITY_INFO,    /**< information */
    OB_LOG_SEVERITY_WARN,    /**< warning */
    OB_LOG_SEVERITY_ERROR, /**< error */
    OB_LOG_SEVERITY_FATAL, /**< fatal error */
    OB_LOG_SEVERITY_OFF       /**< off (close LOG) */
```

```
} OBLogSeverity
```

```
ob::Context    context;
// Set the error level of Log output.
context.setLoggerSeverity(OB_LOG_SEVERITY_DEBUG);
```

This is a global interface. Setting this interface can affect the log level for console output, file output and the log callback function.

If the setLoggerSeverity function is called first to set the log level, and later other interfaces are called to set the log level, the log level will be determined by the last log level set.

7.2 Set the Log Output to the Console

```
ob::Context    context;
// Set the Log output to the terminal console.
context.setLoggerToConsole(OB_LOG_SEVERITY_DEBUG);
```

Note: Only the log level output to the console is affected.

7.3 Set Log Output to File

```
ob::Context    context;
// Set Log output to file
context.setLoggerToFile(OB_LOG_SEVERITY_DEBUG);
```

Note: Only the log level of the output file is affected.

7.4 Set the Log Callback Output

```
#include "libobsensor/hpp/Context.hpp"
// Log output callback function
ob::Context::LogCallback logCallback = [](OBLogSeverity severity, const char *logMsg)
{
    std::cout << "LogSeverity: " << static_cast<int> (severity) << std::endl;
    std::cout << "log: "    << logMsg << std::endl;
};

int main(int argc, char **argv)
{
    ob::Context    context;
    // Set Log output as a callback function to facilitate user management of logs in their own applications.
    context.setLoggerToCallback(OB_LOG_SEVERITY_DEBUG, logCallback);
    ……
}
```

Note: Only the log level output to the callback function is affected.

8 Configuration File Introduction

The following describes Orbbec SDK configuration file (OrbbecSDKConfig_v1.0.xml).

8.1 Log Segment

```
<Log>
<! -Log output level. Valid values: 0-DEBUG, 1-INFO, 2-WARN, 3-ERROR, 4-FATAL, 5-OFF -->
<! -Global log level -->
        <LogLevel>0</LogLevel>
    <! -Log level output to file -->
        <FileLogLevel>0</FileLogLevel>
    <! -Log level output to the console -->
```

```
        <ConsoleLogLevel>1</ConsoleLogLevel>
    <! -Path of the default Log output file: if not configured, Windows/Linux: "./Log"; Android: "/sdcard/Orbbec/Log" -->
      <!--<OutputDir>./log</OutputDir>-->
    <! -The size of the output file by default. Unit: MB -->
        <MaxFileSize>100</MaxFileSize>
    <! -The number of output files by default, which is cyclically overwritten -->
        <MaxFileNum>3</MaxFileNum>
    <! -Asynchronous log output: true-enable, false-disable (default)-->
        <Async>false</Async>
</Log>
```

8.2 Memory Management

```
    <Memory>
        <! -enable the memory pool to avoid memory fragmentation. true-enable, false-disable)-->
        <EnableMemoryPool> true </EnableMemoryPool>
          <! -Set the maximum memory Size of a data frame. Unit: MB. Minimum memory: 100MB -->
        <MaxFrameBufferSize> 2048 </MaxFrameBufferSize>
        <! -The Size of the frame buffer queue in the Pipiline. If you set it to 10, the maximum buffer is 10 frames -->
        <PipelineFrameQueueSize>10</PipelineFrameQueueSize>
        <! -The Size of the frame buffer queue of the internal frame processing unit. If you set it to 10, the maximum buffer is 10 frames.->
        <FrameProcessingBlockQueueSize>10</FrameProcessingBlockQueueSize>
    </Memory>
```

8.3 Pipiline Parameters

```
<Pipeline>
        <Stream>
            <Depth>
              <! -true: uses the default stream configuration of the device field to open the stream. false: uses the specific stream configuration to open the stream. -->
                <UseDefaultStreamProfile>true</UseDefaultStreamProfile>
            </Depth>
            <Color>
          <! -true: uses the default stream configuration of the device field to open the stream. false: uses the specific stream configuration to open the stream. -->
                <UseDefaultStreamProfile>true</UseDefaultStreamProfile>
            </Color>
<! -If you want to enable other streams, please refer to the above configuration for configuration, such as: enable Ir stream -->
      <!--IR    -->
<! -true: uses the default stream configuration of the device field to open the stream. false: uses the specific stream configuration to open the stream. -->
    <UseDefaultStreamProfile>true</UseDefaultStreamProfile>
        <!--/IR -->
            </Stream>
          <! -D2C alignment (0: misalignment; 1: hardware alignment; 2: software alignment) Note: hardware device support is required -->
          <AlignMode>0</AlignMode>
          <! -Pipiline whether to enable frame synchronization when multi-stream is enabled. Note: The hardware device is required to support frame
synchronization.->
            <FrameSync>false</FrameSync>
    </Pipeline>
```

8.4 Device Parameter Configuration

Take Astra+ as an example to describe the device configuration. For more information about the configuration of other modules, see the corresponding module configuration section.

```
<Device>
        <EnumerateNetDevice>false</EnumerateNetDevice>
        <!--The device corresponding to Astra adv is astra+-->
        <AstraAdv>
            <Depth>
                <!--The width of the default resolution, int type-->
                <Width>640</Width>
                <!--The height of the default resolution, int type-->
                <Height>480</Height>
                <!--Default frame rate, int type-->
                <FPS>30</FPS>
```

```
            <!--Default frame format-->
            <Format>Y16</Format>
            <!--Whether to retry after open stream failure, 0 means no retry, >=1 retry and how many times to retry-->
            <StreamFailedRetry>0</StreamFailedRetry>
        </Depth>
        <Color>
            <!--The width of the default resolution, int type-->
            <Width>640</Width>
            <!--The height of the default resolution, int type-->
            <Height>480</Height>
            <!--Default frame rate, int type-->
            <FPS>30</FPS>
            <!--Default frame format-->
            <Format>MJPG</Format>
            <!--Whether to retry after open stream failure, 0 means no retry, >=1 retry and how many times to retry-->
            <StreamFailedRetry>0</StreamFailedRetry>
        </Color>
        <IR>
            <!--The width of the default resolution, int type-->
            <Width>640</Width>
            <!--The height of the default resolution, int type-->
            <Height>480</Height>
            <!--Default frame rate, int type-->
            <FPS>30</FPS>
            <!--Default frame format-->
            <Format>Y16</Format>
            <!--Whether to retry after open stream failure, 0 means no retry, >=1 retry and how many times to retry-->
            <StreamFailedRetry>0</StreamFailedRetry>
        </IR>
    </AstraAdv>
```

9 Exception Handling

When an error occurs during the invocation of the Orbbec SDK interface, it will notify the caller through exceptions. It is essential for developers to handle these exception messages. All interface calls should be wrapped in a try-catch block to handle errors.

```
//C ++ exception handling class: (source code: include/libobsensor/hpp/Error.hpp)
    namespace ob {
    class OB_EXTENSION_API Error {
    public:
        const char *getMessage() const noexcept;
        OBExceptionType getExceptionType() const noexcept;
        const char *getName() const noexcept;
        const char *getArgs() const noexcept;
    };
}
```

Example: build a Context

```
int main(void) {
    try {
        // Create a Context.
        ob::Context ctx;
    catch(ob::Error &e) {
        std::cerr << "function:" << e.getName() << "\nargs:" << e.getArgs() << "\nmessage:" << e.getMessage() << "\ntype:" << e.getExceptionType() << std::endl;
        exit(-1);
    }
    return 0;
}
```

10 End

"The Orbbec SDK C++ Interface Usage Guidelines" will be continuously updated and optimised based on changes in the Orbbec SDK.