wenglor
the innovative family

# Interface Protocol
# **M3DS 3D-Sensoren**




EN

Subject to change without notice
Available as PDF file only
Version 1.0
**www.wenglor.com**

# Table of Contents


**1 Introduction** ..... **3**

**2 System Requirements** ..... **4**

**3 Installation and Folder Structure** ..... **5**

**4 SDK API** ..... **6**

4.1 Device Discovery & Networking ..... 6

4.2 Connection & System Management ..... 7

4.3 Device Configuration ..... 8

4.4 Data Acquisition (Frame Handling) ..... 9

4.5 Data Extraction ..... 13

4.6 Callbacks ..... 16

4.7 Robot Calibration ..... 16

4.7.1 Example Code ..... 22

**5 SDK Error Codes** ..... **24**

**6 SDK ASCII Commands** ..... **25**

6.1 Device Control ..... 25

6.2 Image Control ..... 26

6.3 Acquisition Control ..... 27

6.4 Exposure Control ..... 29

6.5 IO Control ..... 30

6.6 3D Filter Control ..... 31

6.7 User Set Control ..... 33

6.8 Stream Statistics ..... 34

# 1 Introduction

This document describes the functions and the commands for using the library to realize custom application development for the 3D Sensors from wenglor. The library is for users who want to create their own 3D applications using the 3D Sensor series.

# 2 System Requirements

- Operating System: Windows 11 / Ubuntu 22.04
- Processor (minimum): x86_64 ≥ 2.1 GHz, ≥ 4 Cores, AVX2
- RAM (minimum): 8 GB
- Free harddisc space: 1GB
- Network adapter card (minimum): 1 Gbit (recommended 10 Gbit)

Executing any 3D processing application normally requires a powerfull PC. The more resources the PC has, the better and faster processing 3D data is.

# 3 Installation and Folder Structure

Installing the SDK requires administration rights. The SDK installs the following:

- Shared core library (including 3D sensor communication and robot calibration)
- The GenICam Interface
- Documentation PDF and *.md files for AI agents.
- Example code (C++ for the core shared library and Halcon for GenICam)
- C++ Wrapper with example
- Licenses

The installation path as follows:

- Windows: C:/Program Files/wenglor/ Sensor3D SDK
- Linux: /opt/wenglor/sensor3d-sdk

The SDK adds on the Windows PC the following paths:

- Sensor3D_ROOT with the path to the installation folder for cmake configuration
- Add installation folder to PATH environment variable
- Add installation path to GENICAM_GENTL64_PATH

# 4 SDK API

## 4.1 Device Discovery & Networking

| | |
|---|---|
| **Function** | `int Sensor3D_UpdateDeviceList(int* device_count);` |
| **Parameters** | • `device_count` - Pointer to an int that receives the number of devices found. |
| **Description** | Scans the network for available sensors and refreshes the SDK's internal device list. Call this first, then read out each entry with Sensor3D_GetDeviceInfo. |
| **Return** | WE_OK on success.<br>WE_ERR_SOCKET_COMMUNICATION if the network scan itself failed. |

```
int deviceCount = 0;
int res = Sensor3D_UpdateDeviceList(&deviceCount);
if (res != WE_OK) {
    std::cout << "Sensor3D_UpdateDeviceList failed, error code " << res <<
std::endl;
    return -1;
}
if (deviceCount == 0) {
    std::cout << "No sensors found on the network." << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_GetDeviceInfo(int device_index, DeviceInfo* device_info);` |
| **Parameters** | • `device_index` - Index of the device (0-based).<br>• `device_info` - Pointer to a DeviceInfo struct that receives the device's details. |
| **Description** | Retrieves the details of one discovered device by index. Call Sensor3D_UpdateDeviceList first; device_index must be smaller than the device_count it returned. |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_OUT_OF_RANGE if device_index is at or past device_count. |

```
struct DeviceInfo {
    uint32_t struct_size; // set automatically by the constructor
    char serialNumber[64];
    char articleNumber[64];
    char mac[64];
    char ip[64];
    char netmask[64];
    char gateway[64];
    char deviceType[64];
    char deviceState[64];
    char firmwareVersion[64];
    char hardwareVersion[64];
    char userDefinedName[128];
    char connectedTo[64];
    bool authentication;
};
```

```
for (int i = 0; i < deviceCount; ++i) {
    DeviceInfo info{}; // constructor already sets info.struct_size
    if (Sensor3D_GetDeviceInfo(i, &info) != WE_OK) continue;

    std::cout << " [" << i << "] " << info.articleNumber
              << " SN=" << info.serialNumber
              << " IP=" << info.ip << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_SetupDeviceNetwork(const char* articleNumber, const char* serialNumber, const char* ip, const char* mask, const char* gateway, const char* userDefinedName);` |
| **Parameters** | • `articleNumber` - Device article number.<br>• `serialNumber` - Device serial number.<br>• `ip` - New IP address.<br>• `mask` - Subnet mask.<br>• `gateway` - Network gateway.<br>• `userDefinedName` - User defined name for the device. |
| **Description** | Assigns static IP settings and a user defined name to a specific device, identified by its article number and serial number (as returned by Sensor3D_GetDeviceInfo). |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_NULLPTR if any of the string arguments is null.<br>WE_ERR_DATA_NOT_AVAILABLE if no device matching articleNumber/serialNumber was found (run Sensor3D_UpdateDeviceList first). |



**NOTICE**

On success, the device applies the new network settings and reboots immediately to bring them into effect. The current connection to it (if any) is dropped; reconnect at the new ip once the device has come back online.

```
DeviceInfo info{};
Sensor3D_GetDeviceInfo(0, &info); // pick the device to reconfigure
int res = Sensor3D_SetupDeviceNetwork(
    info.articleNumber, info.serialNumber,
    "192.168.100.50", "255.255.255.0", "192.168.100.1",
    "Sensor_Line1_Station3");
if (res == WE_OK) {
    std::cout << "Device network settings updated - device is rebooting." <<
std::endl;
}
```

## 4.2 Connection & System Management

| | |
|---|---|
| **Function** | `SensorHandle Sensor3D_Connect(const char* ip, int timeout);` |
| **Parameters** | • `ip` - Sensor IP address, e.g. "192.168.100.1".<br>• `timeout` - Connection timeout in milliseconds. |
| **Description** | Establishes a TCP/IP connection to the sensor at the given address and returns a handle used by every other API call. |
| **Return** | SensorHandle on success.<br>nullptr if the connection failed (wrong IP, sensor unreachable, or timeout expired). |

```
std::string ip = "192.168.100.1";
SensorHandle sensor = Sensor3D_Connect(ip.c_str(), /*timeout_ms=*/5000);
if (sensor == nullptr) {
    std::cout << "Failed to connect to sensor at " << ip << std::endl;
    return -1;
}
std::cout << "Connected to sensor successfully" << std::endl;
```

| | |
|---|---|
| **Function** | `int Sensor3D_Disconnect(SensorHandle sensor);` |
| **Parameters** | • `sensor` - Active sensor handle to close. |

| | |
|---|---|
| **Description** | Closes the connection to the sensor and releases the handle. The handle must not be used afterwards. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the handle is invalid. |

```
int res = Sensor3D_Disconnect(sensor);
if (res != WE_OK) {
    std::cout << "Failed to disconnect cleanly, error code " << res << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_GetSensorStatus(SensorHandle sensor, int* status);` |
| **Parameters** | • `sensor` - Active sensor handle.<br>• `status` - Pointer to an int that receives the status bitmask. |
| **Description** | Returns the current operational state of the sensor. Currently only SENSOR_CONNECTED (0x1) and SENSOR_DISCONNECTED (0x0) are defined. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the handle is invalid. |

```
int status = 0;
int res = Sensor3D_GetSensorStatus(sensor, &status);
if (res == WE_OK && (status & SENSOR_CONNECTED)) {
    std::cout << "Connected." << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_GetVersion(char* buffer, int size);` |
| **Parameters** | • `buffer` - Target string buffer.<br>• `size` - Size of the buffer in bytes. |
| **Description** | Retrieves the SDK version string. This call does not require a connected sensor. |
| **Return** | WE_OK on success.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer is too small to hold the version string. |

```
char versionBuffer[128] = {};
int res = Sensor3D_GetVersion(versionBuffer, sizeof(versionBuffer));
if (res == WE_OK) {
    std::cout << "SDK version: " << versionBuffer << std::endl;
}
```

## 4.3 Device Configuration

| | |
|---|---|
| **Function** | `int Sensor3D_ReadData(SensorHandle sensor, const char* feature_name, char* buffer, int size);` |
| **Parameters** | • `sensor` - Active sensor handle.<br>• `feature_name` - Name of the feature to read, e.g. "Width".<br>• `buffer` - Target string buffer.<br>• `size` - Size of the buffer. |
| **Description** | Reads a feature/setting by name from the sensor. The result is returned as a string in buffer. |
| **Return** | WE_OK on success.<br>WE_ERR_FEATURE_NOT_FOUND if feature_name is unknown.<br>WE_ERR_FEATURE_NOT_READABLE if the feature exists but cannot be read.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer is too small. |

```
char buffer[256] = {};
int res = Sensor3D_ReadData(sensor, "Width", buffer, sizeof(buffer));
if (res == WE_OK) {
    int width = std::atoi(buffer);
    std::cout << "Width: " << width << std::endl;
} else {
    std::cout << "Failed to read 'Width', error code " << res << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_WriteData(SensorHandle sensor, const char* buffer);` |
| **Parameters** | • `sensor` - Active sensor handle.<br>• `buffer` - Command string to write, e.g. "ExposureTime=15000". |
| **Description** | Writes a feature/setting or executes a command on the sensor by name. Most acquisition/engine settings are stop-mode-only and must be written before AcquisitionStart. |
| **Return** | WE_OK on success.<br>WE_ERR_FEATURE_NOT_FOUND if the feature name is unknown.<br>WE_ERR_FEATURE_NOT_WRITEABLE if the feature is read-only.<br>WE_ERR_ARGUMENT_OUT_OF_RANGE if the value is outside the valid range.<br>WE_ERR_DEVICE_BUSY if the setting cannot be changed while acquisition is running. |

```
int res = Sensor3D_WriteData(sensor, "AcquisitionEngine=1"); // 3D mode
res = Sensor3D_WriteData(sensor, "Engine3D=1"); // phase mode
res = Sensor3D_WriteData(sensor, "ExposureTime=15000"); // microseconds
if (res != WE_OK) {
    std::cout << "Failed to write setting, error code " << res << std::endl;
}
```

## 4.4 Data Acquisition (Frame Handling)

| | |
|---|---|
| **Function** | `int Sensor3D_GetFrame(SensorHandle sensor, FrameHandle* frame, int timeout_ms);` |
| **Parameters** | • `sensor` - Active sensor handle.<br>• `frame` - Pointer to a FrameHandle that receives the new frame.<br>• `timeout_ms` - Maximum time to wait for a frame, in milliseconds. |
| **Description** | Reads out a new data frame from the SDK's internal FIFO buffer. A maximum of 10 frames may be held by the caller simultaneously; once that budget is exhausted the function returns WE_ERR_INSUFFICIENT_MEMORY. Every frame obtained here must eventually be freed with Sensor3D_ReleaseFrame. |
| **Return** | WE_OK on success.<br>WE_ERR_TIMEOUT if no frame arrived in time.<br>WE_ERR_INSUFFICIENT_MEMORY if 10 frames are already outstanding - i.e. 10 frames obtained via Sensor3D_GetFrame (or delivered to a registered frame callback) have not yet been released with Sensor3D_ReleaseFrame. Release older frames to bring the count back down before requesting a new one.<br>WE_ERR_NOT_CONNECTED if the sensor is not connected. |

```
FrameHandle scan = nullptr;
int res = Sensor3D_GetFrame(sensor, &scan, 2000);
if (res != WE_OK) {
    std::cout << "Sensor3D_GetFrame failed, error code " << res << std::endl;
} else {
    // ... extract data from scan, see section 5 ...
    Sensor3D_ReleaseFrame(scan);
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_ReleaseFrame(FrameHandle frame);` |
| **Parameters** | • `frame` - Frame handle to release. |
| **Description** | Frees the memory associated with a frame. Must be called exactly once for every frame returned by Sensor3D_GetFrame or delivered to a frame callback, even if extraction from it failed. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the handle was already released or is invalid. |

```
int res = Sensor3D_ReleaseFrame(scan);
if (res == WE_OK) {
    std::cout << "Frame released, memory is freed" << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetID(FrameHandle frame, unsigned long long* frame_id);` |
| **Parameters** | • `frame` - Active frame.<br>• `frame_id` - Pointer to receive the frame ID. |
| **Description** | Gets the unique sequence ID (picture counter) of the frame. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the frame handle is invalid. |

```
unsigned long long id = 0;
int res = Sensor3D_FrameGetID(scan, &id);
if (res == WE_OK) {
    std::cout << "Received frame #" << id << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetDataSizeBytes(FrameHandle frame, unsigned long long* data_size_bytes);` |
| **Parameters** | • `frame` - Active frame.<br>• `data_size_bytes` - Pointer to receive the size in bytes. |
| **Description** | Returns the total byte size of the frame's raw (GenDC) data. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the frame handle is invalid. |

```
unsigned long long payloadSize = 0;
int res = Sensor3D_FrameGetDataSizeBytes(scan, &payloadSize);
if (res == WE_OK) {
    std::vector<unsigned char> gendcBuffer(payloadSize);
    res = Sensor3D_FrameGetRawData(scan, gendcBuffer.data(), (unsigned int)gend-
cBuffer.size());
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetNrComponents(FrameHandle frame, unsigned int* nr_components);` |
| **Parameters** | • `frame` - Active frame.<br>• `nr_components` - Pointer to receive the component count. |
| **Description** | Returns the number of components (e.g. range, intensity, confidence, chunk) held by the frame. Counts every component, including the metadata/chunk-carrier component - it is the upper bound for component_index in the ...ByIndex functions below. |
| **Return** | WE_OK on success. |

| | |
|---|---|
| | WE_ERR_INVALID_HANDLE if the frame handle is invalid. |

```
unsigned int nrComp = 0;
int res = Sensor3D_FrameGetNrComponents(scan, &nrComp);
if (res == WE_OK) {
    std::cout << "Frame has " << nrComp << " components" << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetComponentInfo(FrameHandle frame, int component_format, ComponentInfo* info);` |
| **Parameters** | • `frame` - Active frame.<br>• `component_format` - Pixel format to look up, e.g. CONFIDENCE8.<br>• `info` - Pointer to a ComponentInfo struct that receives the descriptor. |
| **Description** | Returns a full descriptor for the component matching component_format - geometry, exact data size, and more (see struct ComponentInfo below). Also reports the component's position back in info.index.<br>The metadata/chunk-carrier component is looked up the same way, by its own format (Data8). If component_format matches it, this returns that carrier's descriptor with info.is_chunk == 1, info.width/info.height not meaningful, and info.data_size_bytes covering the whole chunk-carrier blob rather than one chunk. It does not let you look up an individual chunk (e.g. CHUNK_TIMESTAMP) inside that blob - use Sensor3D_FrameGetChunkInfo / Sensor3D_FrameGetChunkInfoByIndex for that. |
| **Return** | WE_OK on success.<br>WE_ERR_DATA_NOT_AVAILABLE if the frame carries no such format. |

```
struct ComponentInfo {
    uint32_t struct_size; // set automatically by the constructor
    uint32_t index;
    uint64_t data_size_bytes;
    uint64_t timestamp;
    int64_t type_id;
    int32_t format;
    uint32_t width;
    uint32_t height;
    uint32_t bits_per_pixel;
    uint32_t region_offset_x;
    uint32_t region_offset_y;
    uint32_t part_count;
    uint16_t padding_x;
    uint16_t padding_y;
    int16_t group_id;
    int16_t source_id;
    int16_t region_id;
    uint8_t is_chunk;
uint8_t is_valid;
    char format_name[64];
};
```

```
ComponentInfo info; // constructor already sets info.struct_size
int res = Sensor3D_FrameGetComponentInfo(scan, CONFIDENCE_8, &info);
if (res == WE_OK) {
    std::cout << info.format_name << " " << info.width << "x" << info.height
                << ", " << info.data_size_bytes << " bytes" << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetComponentInfoByIndex(FrameHandle frame, unsigned int component_index, ComponentInfo* info);` |

| | |
|---|---|
| **Parameters** | • `frame` - Active frame.<br>• `component_index` - Position (0-based, < Sensor3D_FrameGetNrComponents).<br>• `info` - Pointer to a ComponentInfo struct that receives the descriptor. |
| **Description** | Same descriptor as Sensor3D_FrameGetComponentInfo, looked up by position instead of format - use this to enumerate every component in a frame without knowing its formats in advance. |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_OUT_OF_RANGE for an index at or past Sensor3D_FrameGetNrComponents. |

```
unsigned int count = 0;
Sensor3D_FrameGetNrComponents(scan, &count);
for (unsigned int i = 0; i < count; ++i) {
    ComponentInfo info; // constructor already sets info.struct_size
    int res = Sensor3D_FrameGetComponentInfoByIndex(scan, i, &info);
    if (res != WE_OK) continue;
    if (info.is_chunk) continue; // metadata carrier, read via the chunk API instead
    std::cout << "component " << info.index << ": " << info.format_name
              << " " << info.width << "x" << info.height << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetNrChunks(FrameHandle frame, unsigned int* nr_chunks);` |
| **Parameters** | • `frame` - Active frame.<br>• `nr_chunks` - Pointer to receive the chunk count. |
| **Description** | Returns the number of metadata chunks (e.g. timestamp, CRC) carried by the frame. A frame with no metadata component reports 0, not an error. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the frame handle is invalid. |

```
unsigned int nrChunks = 0;
int res = Sensor3D_FrameGetNrChunks(scan, &nrChunks);
if (res == WE_OK) {
    std::cout << "Frame has " << nrChunks << " metadata chunks" << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetChunkInfo(FrameHandle frame, int chunk_id, unsigned int* chunk_index, unsigned int* size_bytes);` |
| **Parameters** | • `frame` - Active frame.<br>• `chunk_id - Chunk ID to look up.`<br>• `chunk_index` - Pointer to receive its position.<br>• `size_bytes` - Pointer to receive its exact size. |
| **Description** | Looks up a chunk by ID (e.g. CHUNK_TIMESTAMP), reporting its position and exact byte size. |
| **Return** | WE_OK on success.<br>WE_ERR_DATA_NOT_AVAILABLE if the frame carries no such chunk. |

```
unsigned int chunkIndex = 0, chunkSize = 0;
int res = Sensor3D_FrameGetChunkInfo(scan, CHUNK_TIMESTAMP, &chunkIndex, &chunkSize);
if (res == WE_OK) {
    std::vector<unsigned char> chunkBuffer(chunkSize);
    res = Sensor3D_FrameGetChunkData(scan, CHUNK_TIMESTAMP, chunkBuffer.data(), chunkSize);
```

}

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetChunkInfoByIndex(FrameHandle frame, unsigned int chunk_index, int* chunk_id, unsigned int* size_bytes);` |
| **Parameters** | • `frame` - Active frame.<br>• `chunk_index` - Position (0-based, < Sensor3D_FrameGetNrChunks).<br>• `chunk_id` - Pointer to receive the chunk's ID.<br>• `size_bytes` - Pointer to receive its exact size. |
| **Description** | Same lookup as Sensor3D_FrameGetChunkInfo, by position instead of ID - use this to enumerate every chunk in a frame without knowing its IDs in advance. |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_OUT_OF_RANGE for an index at or past Sensor3D_FrameGetNrChunks. |

```
unsigned int nrChunks = 0;
Sensor3D_FrameGetNrChunks(scan, &nrChunks);
for (unsigned int i = 0; i < nrChunks; ++i) {
    int chunkId = 0;
    unsigned int chunkSize = 0;
    int res = Sensor3D_FrameGetChunkInfoByIndex(scan, i, &chunkId, &chunkSize);
    if (res != WE_OK) continue;
    std::vector<unsigned char> chunkBuffer(chunkSize);
    res = Sensor3D_FrameGetChunkDataByIndex(scan, i, chunkBuffer.data(), chunk-
Size);
}
```

## 4.5 Data Extraction

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetRange(FrameHandle frame, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame` - Source frame.<br>• `buffer` - Target buffer.<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Copies range data (3D coordinates, POINT3F per point) into buffer. |
| **Return** | WE_OK on success.<br>WE_ERR_DATA_NOT_AVAILABLE if the frame carries no range component.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer_len is smaller than the component's data_size_bytes. |

```
typedef struct POINT3F {
    float x;
    float y;
    float z;
};
```

```
int nrPoints = width * height; // from "Width"/"Height", see Sensor3D_ReadData
std::vector<POINT3F> pointcloud(nrPoints);
int res = Sensor3D_FrameGetRange(scan, (unsigned char*)pointcloud.data(),
                                 (unsigned int)(pointcloud.size() * sizeof(POINT3F)));
if (res == WE_OK) {
    std::cout << "first point = (" << pointcloud[0].x << ", "
              << pointcloud[0].y << ", " << pointcloud[0].z << ")" <<
std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetIntensity(FrameHandle frame, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame` - Source frame.<br>• `buffer - Target buffer.`<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Copies brightness/reflectance (intensity) data into buffer. |
| **Return** | WE_OK on success.<br>WE_ERR_DATA_NOT_AVAILABLE if the frame carries no intensity component.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer_len is too small. |

```
std::vector<uint16_t> image(nrPoints);
int res = Sensor3D_FrameGetIntensity(scan, (unsigned char*)image.data(),
                              (unsigned int)(image.size() * sizeof(uint16_t)));
if (res == WE_OK) {
    // image now holds one intensity value per point
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetComponentData(FrameHandle frame, int component_format, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame - Source frame.`<br>• `component_format - Pixel format of the component to copy.`<br>• `buffer` - Target buffer.<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Copies data for the component matching component_format (e.g. CONFIDENCE_8) into buffer. Use Sensor3D_FrameGetComponentInfo first to size the buffer correctly. |
| **Return** | WE_OK on success.<br>WE_ERR_DATA_NOT_AVAILABLE if the frame carries no such component.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer_len is too small. |

```
std::vector<uint8_t> confidence(nrPoints);
int res = Sensor3D_FrameGetComponentData(scan, CONFIDENCE_8,
                              (unsigned char*)confidence.data(),
                              (unsigned int)(confidence.size() *
sizeof(uint16_t)));
if (res == WE_OK) {
    // confidence copied to user buffer
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetComponentDataByIndex(FrameHandle frame, unsigned int component_index, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame` - Source frame.<br>• `component_index` - Position (0-based, see Sensor3D_FrameGetNrComponents).<br>• `buffer` - Target buffer.<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Copies data for the component matching component_format (e.g. Copies data for the component at a position, instead of a format - pairs with Sensor3D_FrameGetComponentInfoByIndex to enumerate a frame's components without knowing their formats in advance. |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_OUT_OF_RANGE for an index at or past Sensor3D_FrameGetNrComponents.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer_len is too small. |

```
std::vector<unsigned char> buffer(info.data_size_bytes); // info from GetCompo-
nentInfoByIndex
int res = Sensor3D_FrameGetComponentDataByIndex(scan, info.index, buffer.data(),
                            (unsigned int)buffer.size());
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetChunkData(FrameHandle frame, int chunk_id, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame` - Source frame.<br>• `chunk_id` - Metadata ID of the chunk to copy, e.g. CHUNK_TIMESTAMP.<br>• `buffer - Target buffer.`<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Retrieves a metadata chunk (e.g. a timestamp) by ID. |
| **Return** | WE_OK on success.<br>WE_ERR_DATA_NOT_AVAILABLE if the frame carries no such chunk. |

```
uint64_t timestamp = 0;
int res = Sensor3D_FrameGetChunkData(scan, CHUNK_TIMESTAMP,
                            (unsigned char*)&timestamp, sizeof(timestamp));
if (res == WE_OK) {
    std::cout << "Chunk timestamp: " << timestamp << std::endl;
}
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetChunkDataByIndex(FrameHandle frame, unsigned int chunk_index, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame` - Source frame.<br>• `chunk_index` - Position (0-based, see Sensor3D_FrameGetNrChunks).<br>• `buffer - Target buffer.`<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Retrieves a metadata chunk at a position, instead of an ID - pairs with Sensor3D_FrameGetChunkInfoByIndex to enumerate a frame's chunks without knowing their IDs in advance. |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_OUT_OF_RANGE for an index at or past Sensor3D_FrameGetNrChunks. |

```
std::vector<unsigned char> chunkBuffer(chunkSize); // chunkSize from GetChunkIn-
foByIndex
int res = Sensor3D_FrameGetChunkDataByIndex(scan, chunkIndex, chunkBuffer.data(),
chunkSize);
```

| | |
|---|---|
| **Function** | `int Sensor3D_FrameGetRawData(FrameHandle frame, unsigned char* buffer, unsigned int buffer_len);` |
| **Parameters** | • `frame` - Source frame.<br>• `buffer` - Target buffer.<br>• `buffer_len` - Buffer size in bytes. |
| **Description** | Copies the raw (GenDC) container data of the frame into buffer. Size buffer with Sensor3D_FrameGetDataSizeBytes first. |
| **Return** | WE_OK on success.<br>WE_ERR_RETURN_BUFFER_TOO_SMALL if buffer_len is smaller than the value returned by Sensor3D_FrameGetDataSizeBytes. |

```
unsigned long long payloadSize = 0;
Sensor3D_FrameGetDataSizeBytes(scan, &payloadSize);
std::vector<unsigned char> gendcBuffer(payloadSize);
int res = Sensor3D_FrameGetRawData(scan, gendcBuffer.data(), (unsigned int)pay-
loadSize);
if (res == WE_OK) {
    // raw (GenDC) container data copied to user buffer
}
```

## 4.6 Callbacks

| | |
|---|---|
| **Function** | `int Sensor3D_RegisterFrameCallback(SensorHandle sensor, FrameCall-back cb, void* userdata);` |
| **Parameters** | • `sensor` - Active sensor handle.<br>• `cb` - Function pointer of type void(*)(FrameHandle, void*).<br>• `userdata` - User context pointer, passed back unchanged to cb. |
| **Description** | Registers a function to be called on every new frame, as an alternative to polling with Sensor3D_GetFrame. The callback runs synchronously on the SDK's internal receive thread - keep it fast and thread-safe, and release every frame it receives, exactly as with Sensor3D_GetFrame. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the sensor handle is invalid. |

```
static void Scan_CallbackWrapper(FrameHandle scan, void* user_data) {
    if (user_data == nullptr) return;
    UserClass* contextInstance = static_cast<UserClass*>(user_data);
    contextInstance->OnScanReceived(scan);
}
class UserClass {
public:
    void RegisterCallback() {
        int res = Sensor3D_RegisterFrameCallback(sensorHandle, &Scan_Callback-
Wrapper, this);
    }
    void OnScanReceived(FrameHandle &scan) {
        // e.g. process scan data (FrameGetRange, FrameGetIntensity, ...)
        Sensor3D_ReleaseFrame(scan); // free memory
    }
};
```

| | |
|---|---|
| **Function** | `int Sensor3D_UnregisterFrameCallback(SensorHandle sensor);` |
| **Parameters** | • `sensor` - Active sensor handle. |
| **Description** | Removes a previously registered frame callback. After this call, frames are no longer delivered via the callback and must be retrieved with Sensor3D_GetFrame again if needed. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if the sensor handle is invalid. |

```
int res = Sensor3D_UnregisterFrameCallback(sensor);
if (res == WE_OK) {
    std::cout << "Frame callback unregistered" << std::endl;
}
```

## 4.7 Robot Calibration

| | |
|---|---|
| **Function** | `int Sensor3D_InitOnlineCalibration(SensorHandle sensor);` |
| **Parameters** | • `sensor` - Handle to the scanner instance. |

| | |
|---|---|
| **Description** | Initializes a calibration using the connected sensor specified by SensorHandle. |
| **Return** | WE_OK on success.<br>WE_ERR_INVALID_HANDLE if sensor handle invalid. |

| | |
|---|---|
| **Function** | `int Sensor3D_InitOfflineCalibration();` |
| **Description** | Initializes a calibration using offline Data. |
| **Return** | WE_OK on success. |

| | |
|---|---|
| **Function** | `int Sensor3D_SaveCalibrationDataset(const char* folder_path);` |
| **Parameters** | • `folder_path` - Path to the folder where the calibration dataset will be stored. |
| **Description** | Saves the current calibration dataset (poses, associated scans, and configuration) to the specified folder. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_ARGUMENT_INVALID if folder_path is empty.<br>WE_ERR_FILE_WRITE_ERROR if the dataset could not be written. |

| | |
|---|---|
| **Function** | `int Sensor3D_LoadCalibrationDataset(const char* folder_path);` |
| **Parameters** | • `folder_path` - Path to the folder containing the calibration dataset. |
| **Description** | Loads a previously saved calibration dataset from the specified folder and makes it the active dataset for the current calibration session. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_ARGUMENT_INVALID if folder_path is empty or the dataset's sensor id doesn't match the connected sensor.<br>WE_ERR_FILE_READ_ERROR if the dataset folder/file doesn't exist, can't be opened, or is malformed. |

| | |
|---|---|
| **Function** | `int Sensor3D_CaptureCalibrationDataEuler(POSE3D_EULER &pose);` |
| **Parameters** | • `pose` - Robot pose for this scan, using Euler rotation (see POSE3D_EULER structure). |
| **Description** | Captures a new scan from the connected scanner and adds it to the calibration dataset together with the provided robot pose in Euler representation. The function internally acquires the sensor data and links it with the pose. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_INVALID_HANDLE if calibration was initialized offline (no connected sensor).<br>WE_ERR_NOT_CONNECTED if the sensor is not currently connected.<br>WE_ERR_TIMEOUT if a scan could not be captured in time.<br>WE_ERR_DATA_CORRUPTION if the pointcloud/intensity data could not be extracted from the captured frame. |

```
enum AXIS {
    X = 0,
    Y = 1,
    Z = 2
};
struct TRANSLATION {
    TRANSLATION(): x(0), y(0), z(0) {};
    TRANSLATION(float x, float y, float z) :x(x), y(y), z(z) {};
    float x;
```

```
    float y;
    float z;
};
struct ROTATION_EULER{
    ROTATION_EULER(AXIS axis1, AXIS axis2, AXIS axis3): axis_1(axis1),
axis_2(axis2), axis_3(axis3), rot_1(0), rot_2(0), rot_3(0) {};
    ROTATION_EULER(AXIS axis1, AXIS axis2, AXIS axis3, float rot_1, float rot_2,
float rot_3) :axis_1(axis1), axis_2(axis2), axis_3(axis3), rot_1(rot_1),
rot_2(rot_2),rot_3(rot_3) {};
    AXIS axis_1;
    float rot_1;
    AXIS axis_2;
    float rot_2;
    AXIS axis_3;
    float rot_3;
};
struct POSE3D_EULER {
    POSE3D_EULER(AXIS axis1, AXIS axis2, AXIS axis3) : rotation(axis1, axis2,
axis3) {};
    TRANSLATION translation;
    ROTATION_EULER rotation;
};
```

```
//example initialization Z->Y->X
POSE3D_EULER pose(Z, Y, X); // For Euler specify Order of Rotation Axis
pose.translation.x = 100.0f; //mm
pose.translation.y = 200.0f; //mm
pose.translation.z = 300.0f; //mm
pose.rotation.rot_1 = 5.0f; //°
pose.rotation.rot_2 = 3.0f; //°
pose.rotation.rot_3 = 9.0f; //°
// Capture and save scan to calibration dataset
res = Sensor3D_CaptureCalibrationDataEuler(pose);
```

| | |
|---|---|
| **Function** | `int Sensor3D_CaptureCalibrationDataRodrigues(POSE3D_RODRIGUES &pose);` |
| **Parameters** | • `pose` - Robot pose for this scan, using Rodrigues rotation vector (see POSE3D_RODRIGUES structure). |
| **Description** | Captures a new scan from the connected scanner and adds it to the calibration dataset together with the provided robot pose in Rodrigues representation. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_INVALID_HANDLE if calibration was initialized offline (no connected sensor).<br>WE_ERR_NOT_CONNECTED if the sensor is not currently connected.<br>WE_ERR_TIMEOUT if a scan could not be captured in time.<br>WE_ERR_DATA_CORRUPTION if the pointcloud/intensity data could not be extracted from the captured frame. |

```
enum AXIS {
    X = 0,
    Y = 1,
    Z = 2
};
struct TRANSLATION {
    TRANSLATION(): x(0), y(0), z(0) {};
    TRANSLATION(float x, float y, float z) :x(x), y(y), z(z) {};
    float x;
    float y;
    float z;
};
struct ROTATION_RODRIGUES {
    ROTATION_RODRIGUES(): x(0), y(0), z(0) {};
```

```
    ROTATION_RODRIGUES(float x, float y, float z) :x(x), y(y), z(z) {};
    float x;
    float y;
    float z;
};
struct POSE3D_RODRIGUES{
    TRANSLATION translation;
    ROTATION_RODRIGUES rotation;
};
```

```
POSE3D_RODRIGUES pose;
pose.translation.x = 100.0f; //mm
pose.translation.y = 200.0f; //mm
pose.translation.z = 300.0f; //mm
pose.rotation.x = 0.15465939f;
pose.rotation.y = 0.05906952f;
pose.rotation.z = 0.08295479f;
// Capture and save scan to calibration dataset
res = Sensor3D_CaptureCalibrationDataRodrigues(pose);
```

| **Function** | `int Sensor3D_CaptureCalibrationDataQuaternion(POSE3D_QUATERNIONS &pose);` |
|---|---|
| **Parameters** | • `pose` - Robot pose for this scan, using quaternion rotation (see POSE3D_QUATERNIONS structure). |
| **Description** | Captures a new scan from the connected scanner and adds it to the calibration dataset together with the provided robot pose in quaternion representation. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_INVALID_HANDLE if calibration was initialized offline (no connected sensor).<br>WE_ERR_NOT_CONNECTED if the sensor is not currently connected.<br>WE_ERR_TIMEOUT if a scan could not be captured in time.<br>WE_ERR_DATA_CORRUPTION if the pointcloud/intensity data could not be extracted from the captured frame. |

```
enum AXIS {
    X = 0,
    Y = 1,
    Z = 2
};
struct TRANSLATION {
    TRANSLATION(): x(0), y(0), z(0) {};
    TRANSLATION(float x, float y, float z) :x(x), y(y), z(z) {};
    float x;
    float y;
    float z;
};
struct ROTATION_QUATERNIONS {
    ROTATION_QUATERNIONS(): w(0), x(0), y(0), z(0) {};
    ROTATION_QUATERNIONS(float w, float x, float y, float z) :w(w), x(x), y(y),
z(z) {};
    float w;
    float x;
    float y;
    float z;
};
struct POSE3D_QUATERNIONS {
    TRANSLATION translation;
    ROTATION_QUATERNIONS rotation;
};
```

```
POSE3D_QUATERNIONS pose;
```

```
pose.translation.x = 100.0f; //mm
pose.translation.y = 200.0f; //mm
pose.translation.z = 300.0f; //mm
pose.rotation.w = 0.99571678f;
pose.rotation.x = 0.07721926f;
pose.rotation.y = 0.02949258f;
pose.rotation.z = 0.04141816f;
// Capture and save scan to calibration dataset
res = Sensor3D_CaptureCalibrationDataQuaternion(pose);
```

| | |
|---|---|
| **Function** | `int Sensor3D_AddCalibrationDataEuler(const char* img_path, const char* pcl_path, POSE3D_EULER &pose);` |
| **Parameters** | • `img_path` - Path to the image file used for calibration.<br>• `pcl_path` - Path to the pointcloud file used for calibration.<br>• `pose` - Robot pose for this sample, using Euler rotation (see POSE3D_EULER structure). |
| **Description** | Adds an existing calibration sample to the dataset by referencing previously recorded image and pointcloud files, together with the corresponding robot pose in Euler representation. This is intended for offline calibration workflows where scans were recorded beforehand. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_FILE_READ_ERROR if the image or pointcloud file could not be loaded.<br>WE_ERR_CALIBRATION_DATA_INVALID if the image and pointcloud dimensions don't match. |

| | |
|---|---|
| **Function** | `int Sensor3D_AddCalibrationDataRodrigues(const char* img_path, const char* pcl_path, POSE3D_RODRIGUES &pose);` |
| **Parameters** | • `img_path - Path to the image file used for calibration.`<br>• `pcl_path` - Path to the pointcloud file used for calibration.<br>• `pose` - Robot pose for this sample, using Rodrigues rotation vector (see POSE3D_RODRIGUES structure). |
| **Description** | Adds an existing calibration sample to the dataset by referencing previously recorded image and pointcloud files, together with the corresponding robot pose in Rodrigues representation. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_FILE_READ_ERROR if the image or pointcloud file could not be loaded.<br>WE_ERR_CALIBRATION_DATA_INVALID if the image and pointcloud dimensions don't match. |

| | |
|---|---|
| **Function** | `int Sensor3D_AddCalibrationDataQuaternion(const char* img_path, const char* pcl_path, POSE3D_QUATERNIONS &pose);` |
| **Parameters** | • `img_path` - Path to the image file used for calibration.<br>• `pcl_path` - Path to the pointcloud file used for calibration.<br>• `pose` - Robot pose for this sample, using quaternion rotation (see POSE3D_QUATERNIONS structure). |
| **Description** | Adds an existing calibration sample to the dataset by referencing previously recorded image and pointcloud files, together with the corresponding robot pose in quaternion representation. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_FILE_READ_ERROR if the image or pointcloud file could not be loaded. |

| | |
|---|---|
| | WE_ERR_CALIBRATION_DATA_INVALID if the image and pointcloud dimensions don't match. |

| | |
|---|---|
| **Function** | `int Sensor3D_AddCalibrationTarget(const char* target_name);` |
| **Parameters** | • `target_name` - Name or identifier of the calibration target. |
| **Description** | Adds a new calibration target (predefined charuco board config) for the connected scanner. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_ARGUMENT_NULLPTR if target_name is null.<br>WE_ERR_ARGUMENT_INVALID if target_name is not a recognized target. |

```
//target Names and internal defines:
{"zvzj001", mv::calibration::CharucoBoard::charuco_10x8_6000_4666_DICT_4X4_50},
{"zvzj002", mv::calibration::CharucoBoard::charuco_12x9_10000_7777_DICT_4X4_100},
{"zvzj003", mv::calibration::Charu-
coBoard::charuco_15x12_14000_10888_DICT_4X4_100},
{"zvzj004", mv::calibration::Charu-
coBoard::charuco_18x14_18000_14000_DICT_4X4_250},
{"zvzj007", mv::calibration::CharucoBoard::charuco_12x9_30_22_5_DICT_5x5_100},
{"zvzj114", mv::calibration::CharucoBoard::charuco_5x5_35_26_DICT_4x4_50},
{"zvzj115", mv::calibration::CharucoBoard::charuco_16x12_35_26_DICT_4X4_100}
```

| | |
|---|---|
| **Function** | `int Sensor3D_Calibrate3D(const char* save_path, const Calibra-tionType& calib_type);` |
| **Parameters** | • `save_path` - Path to the output folder where the 3D calibration result and log files will be stored.<br>• `calib_type` - CalibrationType to be performed. Eye To Hand or Eye in Hand. |
| **Description** | Performs a 3D hand-eye calibration using the current calibration dataset and the specified calibration configuration.<br>The resulting 4x4 homogeneous transformation and log files are written to save_path. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_ARGUMENT_NULLPTR if save_path is null. |

```
enum CalibrationType {
    EyeToHand,
    EyeInHand
};
```

| | |
|---|---|
| **Function** | `int Sensor3D_Calibrate2D(const char* save_path, const Calibra-tionType& calib_type);` |
| **Parameters** | • `save_path` - Path to the output folder where the 2D calibration result and log files will be stored.<br>• `calib_type` - CalibrationType to be performed. Eye To Hand or Eye in Hand. |
| **Description** | Performs a 2D hand-eye calibration using the current calibration dataset and the specified calibration configuration.<br>The resulting 2D calibration parameters and log files are written to save_path. |
| **Return** | WE_OK on success.<br>WE_ERR_CALIBRATION_NOT_FOUND if no calibration context is initialized.<br>WE_ERR_ARGUMENT_NULLPTR if save_path is null. |

| | |
|---|---|
| **Function** | `int Sensor3D_LoadTransformation(SensorHandle sensor, const char* transformation_path);` |
| **Parameters** | • `sensor` - Handle to the scanner instance.<br>• `transformation_path` - Path to the transformation matrix saved as .json file. |
| **Description** | Loads a 4x4 homogenous transformation matrix from the specified .json file and applies the given transformation to the pointcloud. |
| **Return** | WE_OK on success.<br>WE_ERR_ARGUMENT_NULLPTR if transformation_path is null.<br>WE_ERR_INVALID_HANDLE if sensor handle is invalid. |

```
example.json (mij - i line ,j column) :
{"m00":-0.3344188709906444,"m01":0.7495969309342072,"m02":-0.5711989522955744,"m0
3":1666.0243333258231,"m10":0.9420813259008626,"m11":0.282258398956167,"m12":-0.1
8114465909076277,"m13":-174.33632853563708,"m20":0.025440398316263393,"m21":-0.59
8693742842471,"m22":-0.8005738970738691,"m23":1244.0209075172481,"m30":0.0,"m31":
0.0,"m32":0.0,"m33":1.0}
```

### 4.7.1 Example Code

```
#include <iostream>
#include "Sensor3D/sensor3d.h"
#include "Sensor3D/robot_calibration.h"
#include "Sensor3D/error_codes.h"
#include "Sensor3D/defines.h"
int main() {
    std::cout << "Simple Robot Calibration Example" << std::endl;
    // Connect to sensor
    std::string ip = "192.168.100.1"; // Default IP
    SensorHandle sensorHandle = Sensor3D_Connect(ip.data(), 5000);
    if (!sensorHandle) {
        std::cout << "Failed to connect to sensor at " << ip << std::endl;
        return -1;
    }
    std::cout << "Connected to sensor successfully" << std::endl;
    //Init Calibration
    int res = Sensor3D_InitOnlineCalibration(sensorHandle);
    if (res != WE_OK) {
        std::cout << "Failed initializing calibration" << std::endl;
        return -1;
    }
    // Specify used target
    // See documentation for list of supported targets
std::string target = "zvzj004";
    res = Sensor3D_AddCalibrationTarget(target.data());
if (res != WE_OK) {
        std::cout << "Failed to add calibration target" << std::endl;
        return -1;
    }
    //>>>>>>>>>>>>>> Calibration Loop >>>>>>>>>>>>>>>>>>>>>>>>>>>
    //Repeat for multiple robot poses - the more poses the better
    //[!] try to reach as much translational and rotational variance between sin-
gle poses
    // but keep major parts of the calibration target within the sensors measure-
ment range
    // The readable parameters CalibSetQuality and CalibDof provide an indicator
about the condition
    // of the current calibration dataset based on the entered poses (not analyz-
ing the qualtiy of scan)
    // As a rule of thumb a quality < 2000 (the smaller the better) should be
targeted
    // Dof = 6 (Defined Degrees of Freedom) is required for succesfull calibra-
tion
    //---------------------------------------
// >> >> Move robot to new position >> >>
```

```
    //---------------------------------------
    //---------------------------------------
    // >> >> Read Robot Pose >> >>
    //---------------------------------------
    // Simple pose for demonstration
    POSE3D_EULER pose(Z, Y, X); // For Euler specify Order of Rotation Axis
    pose.translation.x = 100.0f; //mm
    pose.translation.y = 200.0f; //mm
    pose.translation.z = 300.0f; //mm
    pose.rotation.rot_1 = 5.0f; //°
    pose.rotation.rot_2 = 3.0f; //°
    pose.rotation.rot_3 = 9.0f; //°
    // Capture and ADD scan to calibration dataset
    res = Sensor3D_CaptureCalibrationDataEuler(pose);
    if (res == WE_OK) {
        std::cout << "Calibration data captured successfully" << std::endl;
    } else {
        std::cout << "Failed to capture calibration data" << std::endl;
    }
    //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    //Optional: Save Calibration Dataset
    std::string savePath="myCalibrations/2DSet/Data";
    res = Sensor3D_SaveCalibrationDataset(savePath.data());
    if (res == WE_OK) {
        std::cout << "Calibration dataset saved successfully" << std::endl;
    }
    else {
        std::cout << "Failed to save calibration dataset" << std::endl;
    }
    //Execute Calibration on Dataset
    std::string resultPath="myCalibrations/2DSet/Results";
    res = Sensor3D_Calibrate2D(resultPath.data(), EyeToHand);
    if (res == WE_OK) {
        std::cout << "Calibrated successfully" << std::endl;
    }
    else {
        std::cout << "Failed to calibrate see logs for further information" <<
std::endl;
    }
    //[!] if calibration is succesfull the calculated 4x4 homogenous transforma-
tion matrix is saved
    //in the specified result folder as EyeToHand_Result_2D.json respective Eye-
InHand_Result_2D.json
    // both on successfull and non succcesfull calibration additional information
can
    // be derived from the log file "calibFolderName"_CalibrationLog.txt
    std::string resultFileName = resultPath + "/EyeToHand_Result_2D.json";
    //The calculated Transformation can be applied automatically on the sensors
acquired pointclouds
    // with the following function
    res = Sensor3D_LoadTransformation(sensorHandle,resultFileName.data());
    if (res != WE_OK) {
        std::cout << "Failed to Load Transformation" << std::endl;
        return -1;
    }
    return 0;
}
```

# 5 SDK Error Codes

| Code Number | Error Code | Description |
|---|---|---|
| 0 | WE_OK | Operation completed successfully |
| -1 | WE_ERR_INVALID_HANDLE | Handle/reference is invalid or no longer valid |
| -2 | WE_ERR_NOT_CONNECTED | Device or service is not connected |
| -3 | WE_ERR_TIMEOUT | Operation timed out |
| -4 | WE_ERR_UNHANDLED | An unexpected/unhandled error occurred |
| -101 | WE_ERR_ARGUMENT_NULLPTR | A required pointer/argument was null |
| -102 | WE_ERR_ARGUMENT_OUT_OF_RANGE | Argument value is outside the permitted range |
| -103 | WE_ERR_ARGUMENT_INVALID | Argument failed validation |
| -104 | WE_ERR_RETURN_BUFFER_TOO_SMALL | Provided buffer is too small for the result |
| -105 | WE_ERR_FEATURE_NOT_FOUND | Requested feature could not be found |
| -106 | WE_ERR_FEATURE_NOT_WRITEABLE | Feature exists but cannot be modified |
| -107 | WE_ERR_FEATURE_NOT_READABLE | Feature exists but cannot be read |
| -108 | WE_ERR_DEVICE_BUSY | Device is currently busy or locked |
| -109 | WE_ERR_FEATURE_NOT_AVAILABLE | Feature does not exist in current device configuration |
| -201 | WE_ERR_SOCKET_COMMUNICATION | Socket or network communication failed |
| -202 | WE_ERR_DATA_NOT_AVAILABLE | Requested data is not currently available |
| -203 | WE_ERR_NOT_AUTHORIZED | Operation denied due to missing permissions |
| -204 | WE_ERR_PROTOCOL_ERROR | Protocol violation or malformed message |
| -301 | WE_ERR_DATA_STREAM_ERROR | General data stream failure |
| -302 | WE_ERR_DATA_CORRUPTION | Data integrity check failed or data is corrupted |
| -303 | WE_ERR_PROCESSING_FAILED | Internal processing failed |
| -304 | WE_ERR_INSUFFICIENT_MEMORY | Not enough memory to complete the operation |
| -401 | WE_ERR_FILE_NOT_FOUND | File was not found at the specified path |
| -402 | WE_ERR_FILE_ACCESS_DENIED | Insufficient permissions to access the file |
| -403 | WE_ERR_FILE_WRITE_ERROR | Failed to write data to file |
| -404 | WE_ERR_FILE_READ_ERROR | Failed to read data from file |
| -405 | WE_ERR_FILE_FORMAT_INVALID | File format is unsupported or invalid |
| -501 | WE_ERR_CALIBRATION_FAILED | Calibration process did not complete successfully |
| -502 | WE_ERR_CALIBRATION_DATA_INVALID | Calibration data is corrupt, incomplete, or incompatible |
| -503 | WE_ERR_CALIBRATION_NOT_FOUND | Required calibration profile or file is missing |

# 6 SDK ASCII Commands

**How to use commands**

Commands that have access mode Write → use the M3DS SDK API function Sensor3D_WriteData

Commands that have access mode Read → use the M3DS SDK API function Sensor3D_ReadData

See Device Configuration [▶ 8]

**See also**

SDK API [▶ 6]

## 6.1 Device Control

| | |
|---|---|
| **Command** | DeviceModelName, DeviceSerialNumber, DeviceFirmwareVersion, DeviceVersion, DeviceVendorName, DeviceHwID |
| **Access Mode** | Read only |
| **Type** | string |
| **Description** | Returns the article number, serial number, FW version, version (hardware version), the vendor name and the hardware ID of the connected device. |

| | |
|---|---|
| **Command** | SensorWidth, SensorHeight |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Returns the width and the height of the camera chip in the device |

| | |
|---|---|
| **Command** | IntrinsicCameraParameters |
| **Access Mode** | Read only |
| **Type** | string |
| **Description** | Returns the intrinsic & extrinsic camera parameters of the device. See examples below. IntrinsicCameraParameters: {"alpha":-1.11569,"focal_length_x":8134.54,"focal_length_y":8124.53,"k1":-0.0252584,"k2":0.159551,"k3":-1.82418,"optical_center_x":2041.34,"optical_center_y":1454.26,"p1":-8.94306e-05,"p2":0.000370619} ExtrinsicCameraParameters {"m00":0.999997,"m01":-0.00243729,"m02":-0.000457324,"m03":0.111022,"m10":0.00243895,"m11":0.99999,"m12":0.00368011,"m13":0.138751,"m20":0.00044835,"m21":-0.00368122,"m22":0.999993,"m23":37.4106,"m30":0.0,"m31":0.0,"m32":0.0,"m33":1.0} |

| | |
|---|---|
| **Command** | Scan3dAxisMinStartA, Scan3dAxisMinStartB, Scan3dAxisMaxStartA, Scan3dAxisMaxStartB |
| **Access Mode** | Read only |
| **Type** | string |
| **Description** | Returns the start working range in X (A), Y (B) coordinates of the connected device. |

| | |
|---|---|
| **Command** | Scan3dAxisMinA, Scan3dAxisMinB, Scan3dAxisMaxA, Scan3dAxisMaxB |
| **Access Mode** | Read only |
| **Type** | string |
| **Description** | Returns the end working range in X (A), Y (B) coordinates of the connected device. |

| | |
|---|---|
| **Command** | Scan3dAxisMinC, Scan3dAxisMaxC |
| **Access Mode** | Read only |
| **Type** | string |
| **Description** | Returns the end working range in Z (C) of the connected device |

The working range of the device is represented as a truncated pyramid. Scan3dAxisMinStartA, Scan3dAxisMinStartB, Scan3dAxisMaxStartA, Scan3dAxisMaxStartB represent the plane of the upper part (near the sensor) of the truncated pyramid. Scan3dAxisMinA, Scan3dAxisMinB, Scan3dAxisMaxA, Scan3dAxisMaxB represent the button part (away from the senor) of the truncated pyramid. Scan3dAxisMinC, Scan3dAxisMaxC is the distance between the upper and the button part of the planes.

| | |
|---|---|
| **Command** | TemperatureInternalHousing, TemperatureLED, TemperatureImageSensor, TemperatureLightEngine, TemperatureCPU |
| **Access Mode** | Read only |
| **Type** | float |
| **Description** | Returns the temperature of the internal housing, LED, image sensor, Light Engine and CPU of the device |

## 6.2 Image Control

| | |
|---|---|
| **Command** | Width, Height |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Returns the actual Width and Height of the camera. If Decimation = 1; the values Width & Height are equal to SensorWidth and SensorHeight If Decimation = 2, the Width & Height values are halved. |
| **Dependencies** | The value will be updated if one of those commands are written to the device Decimation. |

| | |
|---|---|
| **Command** | ComponentEnableConfidence |
| **Access Mode** | Read/Write |
| **Type** | boolean |
| **Description** | Enable/disable the Confidence component in the 3D mode of the device. |
| **Availability** | During stop mode |

| | |
|---|---|
| **Command** | Decimation |
| **Access Mode** | Read/write |
| **Type** | integer |
| **Value** | 1,2 |
| **Description** | Setup the vertical and horizontal decimation of the camera chip 1: no decimation 2: reduces the vertical and horizontal resolution of the image by the specified vertical decimation factor. |
| **Dependencies** | The value will result in invalidating the Width and the Height Decimation. |
| **Availability** | Dsuring stop mode |

| | |
|---|---|
| **Command** | PayloadSize |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Returns the total number of bytes of the frame container. |

| | |
|---|---|
| **Dependencies** | The value will be updated if one of those commands are written to the device AcquisitionEngine ComponentEnableConfidence Decimation. |

## 6.3 Acquisition Control

| | |
|---|---|
| **Command** | AcquisitionMode |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Description** | 0: sets the device into continuous mode<br>1: sets the device into single frame mode<br>In single frame mode, after the client sends the first AcquisitionStart command, the device locks its resources and sends back only one frame. The client has to send another command AcquisitionStart to get a new frame or send AcquisitionStop to unlock the resources in the device In continuous, after the client sends the first AcquisitionStart, the device locks its resources and sends continuously frames until the client sends the command AcquisitionStop. This feature is locked during acquisition. |
| **Availability** | During stop mode |

| | |
|---|---|
| **Command** | AcquisitionStart |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Initialize the device to start the acquisition mode. In acquisition mode, the device sends frames (depending trigger mode, trigger source and acquisition mode of the device) There will be commands that are not allowed to be written to the device during the acquisition. This is mentioned in the availability in every command. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | AcquisitionStop |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Stops the acquisition mode on the device and frees the resources of the device. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | AcquisitionEngine |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1 |
| **Description** | 0: sets the device into 2D mode<br>1: sets he device into 3D mode.<br>In 2D mode, the device sends only Intensity component. In 3D mode, the device sends Range, Intensity and Confidence components. it is possible to disable Confidence component in 3D mode, see ComponentEnableConfidence. |
| **Dependencies** | Setting the device to 2D mode will make 3D related commands not available Engine3D StackSize AcquisitionFrameRate AcquisitionPriority all Filters in 3D setting the device to 3D mode will make 2D related commands not available Projection2D. |
| **Availability** | Only in stop mode |

| | |
|---|---|
| **Command** | Engine3D |

| | |
|---|---|
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1 |
| **Description** | 0: stripe<br>1: phase<br>This feature defines the pattern of the light engine for the 3D mode. In stripe mode, the sensor provides a fast reconstruction with good quality even on glossy or reflective objects. It is particularly suitable for applications where short exposure times are important. In phase mode, the sensor generates highly precise point clouds with maximum accuracy requiring slightly longer acquisition times. It is particularly suitable for matte or diffuse surfaces. |
| **Availability** | Only in stop mode |

| | |
|---|---|
| **Command** | StackSize |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 14,23,26 |
| **Description** | It sets the number of images in the stack generating the point cloud. if Engine3D=0 (stripe) only stack size 23 is available if Engine3D=1 (phase) stack size 14 & 26 is available. |
| **Dependencies** | This value will be updated when Engine3D changes its value |
| **Availability** | Only in stop mode |

| | |
|---|---|
| **Command** | AcquisitionFrameRate |
| **Access Mode** | Read/Write |
| **Type** | float |
| **Value** | 0.1 .... 4.1 [Hz] |
| **Description** | Sets the acquisition frame rate on the device in 3D mode. |
| **Dependencies** | The value will be updated when one the following features changes its value: Decimation, AcquisitionEngine, ExposureTime, Engine3D & StackSize. |
| **Availability** | Only in stop mode only in 3D mode & TriggerSource is internal (0) |

| | |
|---|---|
| **Command** | AcquisitionPriority |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1,2 |
| **Description** | 0: individual<br>1: accurate<br>2: fast (available only in Engine3D = Phase)<br>The feature provides the user with an easy way to configure the device based on the quality of the generated point cloud by controlling internally both StackSize and Decimation. The user can select between high quality (accurate) point cloud or less quality point cloud but favouring fast acquisition (fast). The user can also select individual to setup his own setting. |
| **Dependencies** | The value will be updated when StackSize Decimation change their values. |
| **Availability** | Only in stop mode only in 3D mode |

| | |
|---|---|
| **Command** | Projection2D |
| **Access Mode** | Read/Write |

| | |
|---|---|
| **Type** | integer |
| **Value** | 0,1 |
| **Description** | 0: solid<br>1: cross hair sets the projector's pattern in 2D mode |
| **Availability** | In stop mode in 2D mode |

| | |
|---|---|
| **Command** | TriggerSource |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1,2,3,4,5 |
| **Description** | Set the trigger source of the device<br>0: internal trigger, the device will scan in a frequency as defined by Acquisition-FrameRate<br>1: software, the device will be triggered by the command SoftwareTrigger<br>2..5: Line1...Line4, the device will be triggered per hardware signal. |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | SoftwareTrigger |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Trigger the device when TriggerSouce is set to software. |

## 6.4 Exposure Control

| | |
|---|---|
| **Command** | ExposureTime |
| **Access Mode** | Read/Write |
| **Type** | float |
| **Value** | min ..... 80527 [µs] |
| **Description** | Sets the device's exposure time. min Value depends on the value of Engine3D; min = 1000 if Engine3D = stripe min = 11800 if Engine3D = phase. |
| **Dependencies** | The value will be updated when Engine3D changes its value. |
| **Availability** | Only in stop mode |

| | |
|---|---|
| **Command** | Gain |
| **Access Mode** | Read/Write |
| **Type** | float |
| **Value** | 0 .... 100 |
| **Description** | Sets the gain of the camera chip. |
| **Dependencies** | The values will be updated when Auto Adjust is executed. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | LightBrightness |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0 .... 100 |
| **Description** | Sets the light brightness of the projector. |
| **Dependencies** | The values will be updated when Auto Adjust is executed |

| | |
|---|---|
| **Availability** | Always |

| | |
|---|---|
| **Command** | AutoAdjust |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Activates the auto adjust process on the device. Auto Adjust computes the optimal values for ExposureTime, Gain and Lightbrightness for the selected region of the scene, see the commands AutoAdjustAOI* below. Once the operation is done, the Auto Adjust then updates the values of the ExposureTime, Gain and LightBrightness. It can be that the algorithm will fail to compute the optimal values; in this case the Sensor3D_WriteData function will return an error. The command might take up to 5 seconds time to compute the values, during this time, the device will flash several times to capture 2D intensity images to compute the best results. |
| **Dependencies** | A success run of Auto Adjust will result in updating the values of ExposureTime Gain LightBrightness. |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | AutoAdjustAOIWidth, AutoAdjustAOIOffsetX, AutoAdjustAOIHeight, AutoAdjustAOIOffsetY |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | see the description below |
| **Description** | Sets the Area of Interest for the Auto Adjust algorithm to compute the optimal values of ExposureTime, Gain and Lightbrightness. The values of Width, Height, OffsetX and OffsetY should fulfill the following conditions: AOI Width + AOI OffsetX <= Width AOI Height + AOI OffsetY <= Height where Width and Height values are device type dependent, see commands Width, Height. |
| **Dependencies** | The values will be updated when Decimation changes its value. |
| **Availability** | In stop mode |

## 6.5 IO Control

| | |
|---|---|
| **Command** | EA1LineMode, EA2LineMode, EA3LineMode, EA4LineMode |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1 |
| **Description** | Set the Line to be input or output.<br>0: input<br>1: output |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | EA1LineSource, EA2LineSource, EA3LineSource, EA4LineSource |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1,2 |
| **Description** | Set the Line source when it is output.<br>0: Frame active<br>1: Sensor busy<br>2: Timer0 |

| | |
|---|---|
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | Timer0Source |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1 |
| **Description** | Set the source of triggering an internal timer.<br>0: frame start<br>1: frame end |
| **Availability** | In stop mode and when one of the line source of at least one Line is set to output and the LineSource is set to Timer0. |

| | |
|---|---|
| **Command** | Timer0Duration |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0 ... 80527 [µs] |
| **Description** | Set the duration of the Timer0 Pulse. |
| **Availability** | In stop mode and when one of the line source of at least one Line is set to output and the LineSource is set to Timer0. |

| | |
|---|---|
| **Command** | Timer0Delay |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0 ... 80527 [µs] |
| **Description** | Set the delay of triggering the timer |
| **Availability** | In stop mode and when one of the line source of at least one Line is set to output and the LineSource is set to Timer0. |

## 6.6 3D Filter Control

| | |
|---|---|
| **Command** | MedianFilter |
| **Access Mode** | Read/Write |
| **Type** | boolean |
| **Value** | 0/1 |
| **Description** | Enable/disable median filter on point cloud. |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | SmoothingFilter |
| **Access Mode** | Read/Write |
| **Type** | boolean |
| **Value** | 0/1 |
| **Description** | Enable/disable smoothing filter of point cloud |
| **Availability** | Always |

| | |
|---|---|
| **Command** | SmoothingFilterKernel |
| **Access Mode** | Read/Write |
| **Type** | integer |

| | |
|---|---|
| **Value** | 3,5,7,9 |
| **Description** | Defines the kernel of the smoothing filter, higher value will result in a smoothed surface of the point cloud, but also leads to remove some sharp aspects of the surface. higher kernel value will result in higher processing time of the filter. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | OutlierFilter |
| **Access Mode** | Read/Write |
| **Type** | boolean |
| **Value** | 0/1 |
| **Description** | Enable/disable outlier filter, see the description below. |

| | |
|---|---|
| **Command** | OutlierFilterSigma |
| **Access Mode** | Read/Write |
| **Type** | float |
| **Value** | 1.0 ... 4.0 |
| **Description** | Sets the sigma property of the filter, see the description below. |

| | |
|---|---|
| **Command** | SetOutlierFilterMinCluster |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0 ... 100000 |
| **Description** | Defines the minimum cluster size for the filter to remove, see the description below |

Outlier Filter consists of 2 filtering mechanisms.

1. The distance of a point in regard to its neighbours.
2. The number of points inside a connected region of points. Whereby the belonging of a point to a certain region is defined by the distance to his neighbouring point inside this region.

The first mechanism is determined by the sigma value. The algorithm computes the standard deviation for the distance of all points to their direct neighbours. The sigma value is a relative threshold as a multiple of this computed standard deviation. If a points distance to its neighbours is bigger than sigma*standardDeviation it is considered an outlier.

Assuming Points are normally distributed this would mean:

- 68% of the data falls within one standard deviation of the mean.
- 95% of the data falls within two standard deviations of the mean.
- 99.7% of the data falls within three standard deviations of the mean.

According to this a SigmaValue of 1 would mean 32% of all points are filtered out. (5% for 2σ, 0.3% for 3σ respectively)

The cluster mechanism aims to reject whole regions of points which are grouped together by their spatial proximity. The minimum cluster size threshold thereby determines the minimum number of points under which the whole cluster should be neglected.

In conclusion the sigma value targets sparse outliers which stand out by the distance to their direct neighbours. The MinClusterSize threshold in contrast is meant to filter out artifacts which build groups of close points.

| | |
|---|---|
| **Command** | WorkingAreaCFilter |
| **Access Mode** | Read/Write |
| **Type** | boolean |

| | |
|---|---|
| **Value** | 0/1 |
| **Description** | Enable/disable filtering out points that are located outside the defined working range limits in C (Z) coordinate, see WorkingAreaCMin, WorkingAreaCMax. |

| | |
|---|---|
| **Command** | WorkingAreaCMin, WorkingAreaCMax |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | min working range of device in C (Z) coordinate<br>max working range of device in C (Z) coordinate. |
| **Description** | Defines the limits of the filter in C coordinate. The WorkingAreaCMax should be always larger than the WorkingAreaCMin. |

| | |
|---|---|
| **Command** | ExtendedWorkingArea |
| **Access Mode** | Read/Write |
| **Type** | boolean |
| **Value** | 0/1 |
| **Description** | Enable/disable extended working range. This shows/hide points outside the standard working range. The point outside working range are not guaranteed. The results might be different across different devices having the same article number. |

| | |
|---|---|
| **Command** | ConfidenceFilter |
| **Access Mode** | Read/Write |
| **Type** | boolean |
| **Value** | 0/1 |
| **Description** | Enable/disable filtering out point cloud based on ConfidenceThreshold. |

| | |
|---|---|
| **Command** | ConfidenceThreshold |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0 ... 255 |
| **Description** | Set the threshold for the confidence filter |

## 6.7 User Set Control

| | |
|---|---|
| **Command** | UserSetDefault |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Value** | 0,1,2,3,4 |
| **Description** | Select the user set to be loaded when the client connects to the device<br>0: Default<br>1: Dark<br>2: Reflective<br>3: Translucent<br>4: Accurate |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | UserSetCurrent |

| | |
|---|---|
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | returns the last loaded user set |

| | |
|---|---|
| **Command** | UserSetLoad |
| **Access Mode** | Write only |
| **Type** | integer |
| **Value** | 0,1,2,3,4 |
| **Description** | Loads the user set<br>0: Default<br>1: Dark<br>2: Reflective<br>3: Translucent<br>4: Accurate |
| **Dependencies** | Loading a user set will result in updating all features. |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | UserSetSave |
| **Access Mode** | Write only |
| **Type** | integer |
| **Value** | 0,1,2,3,4 |
| **Description** | Saves the current device configurations into a user set<br>0: Default<br>1: Dark<br>2: Reflective<br>3: Translucent<br>4: Accurate |
| **Availability** | In stop mode |

| | |
|---|---|
| **Command** | DeviceReset |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Calling DeviceReset overwrites the UserSets saved on the device to their defaults defined by the installed FW. Then it loads the Userset0 (Default) to the device. |
| **Availability** | During stop mode |

## 6.8 Stream Statistics

Stream statistics provide real-time insight into the SDK's data receive and processing pipeline. All read-only counters are available at any time; use ResetStreamStatistics to zero them.

| | |
|---|---|
| **Command** | StreamReceiveFrequency |
| **Access Mode** | Read only |
| **Type** | float |
| **Description** | Frames per second entering the SDK receive pipeline. |
| **Availability** | During acquisition |

| | |
|---|---|
| **Command** | StreamReceiveBitrate |

| | |
|---|---|
| **Access Mode** | Read only |
| **Type** | float |
| **Description** | Received data load in Mbit/s. |
| **Availability** | During acquisition |

| | |
|---|---|
| **Command** | StreamProcessFrequency |
| **Access Mode** | Read only |
| **Type** | float |
| **Description** | Frames per second fully processed by the SDK |
| **Availability** | During acquisition |

| | |
|---|---|
| **Command** | LostFramesSDK |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Number of frames dropped by the SDK frame queue (queue overflow) Saturates at 2147483647. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | LostFramesDevice |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Number of frames lost on the device/network side before reaching the SDK Saturates at 2147483647. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | InputQueueCounter |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Current number of frames waiting in the SDK input frame queue. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | LostFramesTotal |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Total frames lost (SDK dropped + device/network lost combined) Saturates at 2147483647. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | InputQueueSize |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Default** | 2 |
| **Value** | 1 ... 20 |
| **Description** | Maximum number of frames the SDK input frame queue can hold. Values outside 1 ... 20 are rejected. Takes effect immediately; shrinking the queue trims the oldest buffered frames. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | OutputQueueSize |
| **Access Mode** | Read/Write |
| **Type** | integer |
| **Default** | 2 |
| **Description** | Maximum number of processed scans the SDK output queue (the FIFO delivered to the client) can hold. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | OutputQueueCounter |
| **Access Mode** | Read only |
| **Type** | integer |
| **Description** | Current number of processed scans waiting in the SDK output queue. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | ResetOutputQueue |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Clears the SDK output queue. |
| **Availability** | Always |

| | |
|---|---|
| **Command** | ResetStreamStatistics |
| **Access Mode** | Write only |
| **Type** | command |
| **Description** | Resets all stream statistics counters (dropped frames, lost frames) to zero. |
| **Availability** | Always |