SICK
Sensor Intelligence.

# OD7000Lib

Support Library for OD7000 Device Connections

## Application Programming Interface Reference

Version 4.2.9.0 (created on 2021-09-21)

2021 • EN • v4.2

9403118/2024-06-18 · Änderungen vorbehalten · SICK AG · Waldkirch · Germany · www.sick.com

# Contents

9403118/2024-06-18 · Änderungen vorbehalten · SICK AG · Waldkirch · Germany · www.sick.com

# Part I

# Introduction

# Chapter 1

# Introduction

## 1.1 Quick Start

**Connect** As a starting point, to connect to a sensor, simply use the OpenConnection() function. It takes four arguments:

1. the connection specification, e.g. `COM15` or `/dev/ttyS0` for a serial connection on Windows and Linux respectively, or `192.168.170.2` for TCP/IP-based network connections[3]
2. the device type given as an integer enumeration
3. the device buffer size, i. e. the size of an internal circular buffer for temporary device output storage (defaulting to 32 MB when 0 is passed)
4. the pointer to store a connection handle

Upon successful connection establishment, a connection handle is assigned which will be used later for communication (samples, sensor control commands).

**Select Signals** After the connection is established, the next step is to select the measurement signals to acquire. This is done by creating an output signal command, adding required signals to the command and executing the command.

**Receive Data** Generally, the library first stores all device output in an internal circular buffer. From here, GetNextSamples() will read a number of samples into a user-visible data sample buffer. The number of samples is limited by a user-defined maximum and a time-out, whichever is reached first. GetLastSample(), on the other hand, will read the latest data and fast-forward the read pointer to past the last received telegram. The address of the data sample buffer will be indicated by the GetNextSamples()/ GetLastSample() function. If the circular buffer is not being read for a longer period, the internal buffer may overflow and hence the functions return an error. In this case, simply call the function FlushConnectionBuffer().

**Buffer Data** It is also possible to pass a user buffer to the library which will be filled automatically with multiple samples. The relevant function for this scenario is ActivateAutoBufferMode().

**Disconnect** Finally, CloseConnection() will close the connection.

NOTE: The above approach of “Quick Start” uses the synchronous mode, i. e. “send commands and wait for response”. It means if a command is sent to the sensor, all the data samples already received will be ignored and the library waits for the response (*synchronous mode*).

[3] Currently OD7000Lib is limited to IPv4.

9403118/2024-06-18 · Änderungen vorbehalten · SICK AG · Waldkirch · Germany · www.sick.com

# Chapter 2

# Different modes of operation

Operating a OD7000 sensor has two main aspects:

- Controlling the device, i.e. sending commands which query or modify its status
- Acquiring measurement data

There are two different ways of accomplishing these tasks, the *synchronous* and the *asynchronous* mode.

## 2.1 Synchronous mode

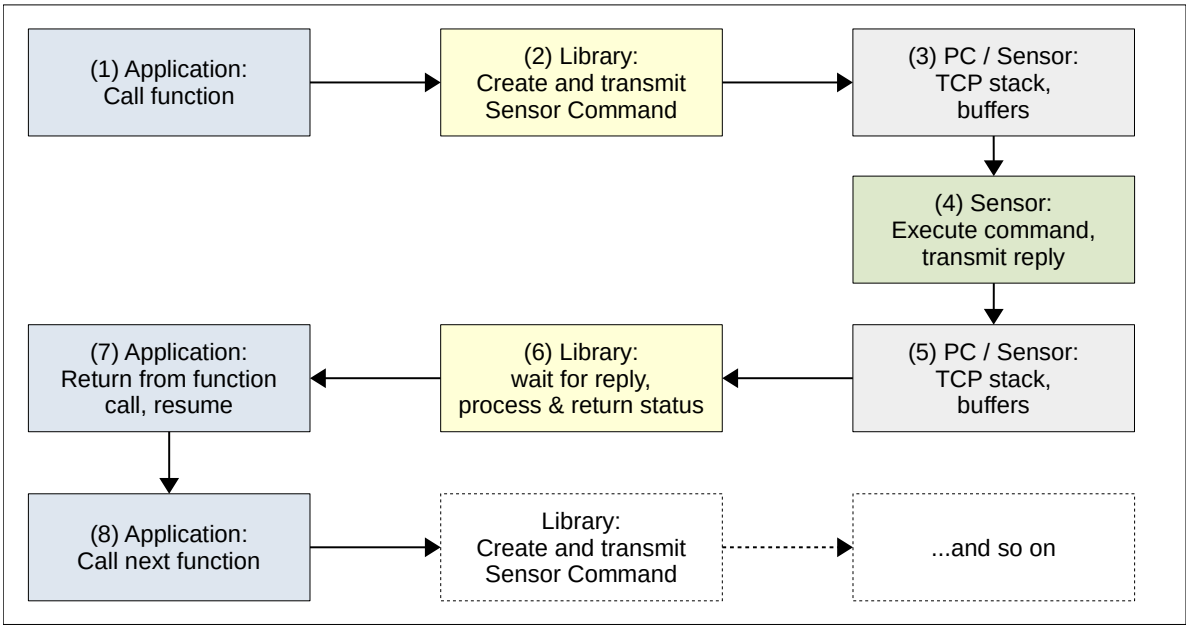


Figure 2.1: The most straight forward approach: Synchronous sensor control

**Execute command** When using the synchronous mode ExecCommand() function, the function call typically[1] causes a device command to be sent to the OD7000 device. The OD7000 device will then process the received command and respond. The response will in turn be further processed by the library to form result codes for the caller. This sequence is illustrated in figure 2.1.

After making an API function call, the caller checks the result code for potential errors and further processes the values passed back by the library. If no error occurred, the caller can safely assume that the command has successfully completed and the next one can be issued.

In *synchronous mode*, the function call for executing command will thus not return until the command has was created, sent, processed and responded to by the device. That is, the call will block the caller until it is done. Any sample data produced by the sensor in the meantime will be ignored and skipped.

[1] There are cases in which a single command issued from the API results in several device commands.

**Collect Data** Measurement data is typically obtained by calling GetNextSamples(). This function does not block and returns either the required number of samples or the available number. One can also use the *auto buffer save* functionality (see ActivateAutoBufferMode()) which runs in the background, moving data into a user buffer. Note, however, during the period in which data is automatically stored into the user buffer, you **should not** send any commands to the device as this has the potential to cause inconsistencies in the saved data[2].

**Pros & Cons** The *synchronous mode* can thus be summarized as follows:

+ sending a command is a strictly serial and atomic operation
+ the mode is simple and safe
– batch command processing (as during initialization) is slow because only one single command is sent and processed at a time

A general pattern for writing software that runs in synchronous mode:

- Open connection
- Set up device parameters with ExecCommand() or convenient functions/wrapper classes
- Collect data with GetNextSamples() or ActivateAutoBufferMode()

## 2.2 Asynchronous Mode

**What for?** In many usage scenarios, the synchronous mode above will be a perfect fit. Initialization of the sensor is often a one-time task.

If your application falls into this category, then you can skip the rest of this chapter and read the sections covering GetNextSamples(), ActivateAutoBufferMode(), and so on.

If, on the other hand, device reconfiguration happens frequently in your application, the time to issue multiple discrete commands and wait for their respective responses may become significant. The *asynchronous mode* sidesteps the bottleneck of strict serialization by enabling you to issue commands without waiting for the completion of previous ones.

Or, if you need to design an interactive application that helps the user to set up some measurement, goals may be to:

- adjust the probe while observing a live spectrum
- see how peak height values are changing in response to measurement frequency or light intensity adjustments
- see how noise levels vary with different averaging or median settings

Synchronous mode would not only cause the loss of data samples but might also degrade the application's responsiveness (since synchronous command execution is blocking). In this case, you should consider the *asynchronous mode*.

**Process Device Output** In synchronous mode, synchronous functions, like ExecCommand(), GetNextSamples(), trigger the processing of OD7000 device output (stored in the internal circular buffer). In asynchronous mode, ExecCommandAsync() is used for submitting commands - with the key difference that it returns immediately, leaving device output processing to some other function to be called later: ProcessDeviceOutput().

[2] In synchronous mode it is even forbidden to do so.

`ProcessDeviceOutput` collects all device output and wraps it into data samples or command responses / updates.

An application may actively call this function e. g. in a GUI timer routine (i. e. in the same thread as the `ExecCommandAsync()` call). Then a button handler may initiate a device command while the timer routine updates / refreshes the dialog state.

Alternatively, the host application can request OD7000Lib to take care of this task by calling StartAutomaticDeviceOutputProcessing(). `StartAutomaticDeviceOutputProcessing` actually initializes an extra internal thread to call ProcessDeviceOutput() constantly. In both

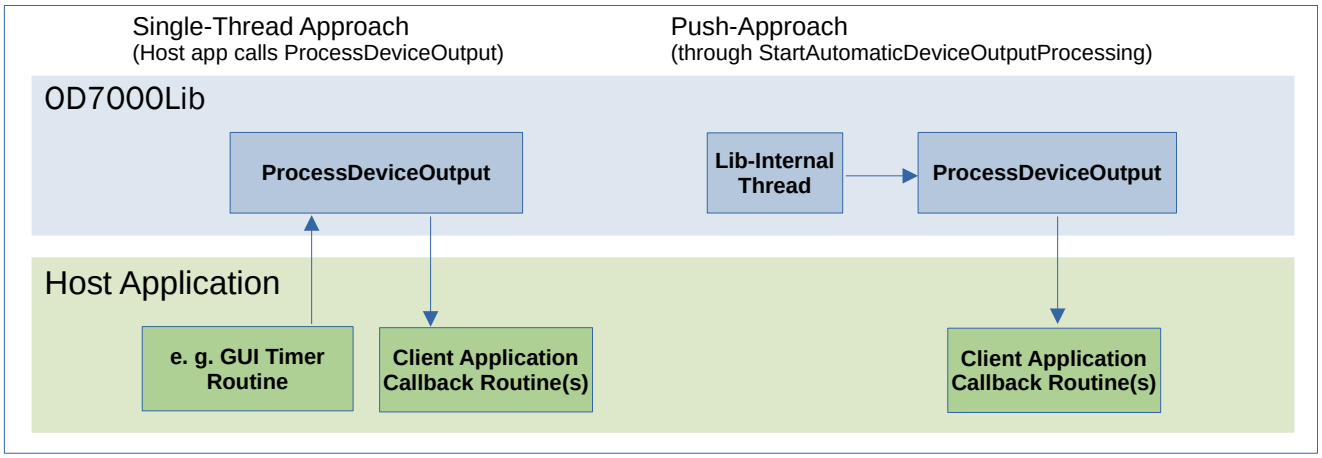


Figure 2.2: `ProcessDeviceOutput` can be called either from the host application or a thread inside OD7000Lib.

As these functions are invoked from inside ProcessDeviceOutput() (i. e. it's the client itself which eventually calls it's own callback routine), they also run in the same thread as the ProcessDeviceOutput() call(see figure 2.3).

A single-threaded approach has some noticeable advantages:

- being single-threaded, it is easy and safe to implement, no specific locking / synchronization measures need to be taken.
- given reasonable load, it typically results in quite responsive applications.

However, the main downside is poor scalability in terms of data bandwidth / load since it only works well if the GUI timer routine does not take significant time to execute.

In general, leveraging the power of modern multi-core PCs requires a multi-thread approach. For more details on the library's thread model, please refer to chapter 3.

**Command Callback Function** As mentioned above, the command responses are forwarded through callback functions (see figure 2.4). Callback functions can be registered for specific command responses (directly in ExecCommandAsync()) as well as for all the responses/updates in RegisterGeneralResponseAndUpdateCallback().

The callback function which is registered through RegisterGeneralResponseAndUpdateCallback() will be invoked upon receiving any responses/updates from the device, hence the name "general callback function". Due to the nature of command updates as "spontaneous" messages from the device without prior command, a general callback function is the only way of receiving such updates. Normally the application code should set a "general callback function" right after opening a connection.

The callback function which is passed into ExecCommandAsync() will, on the other hand, only apply to the command in this specific ExecCommandAsync() call. If "general callback function" has been previously registered, the callback function in ExecCommandAsync() will be called

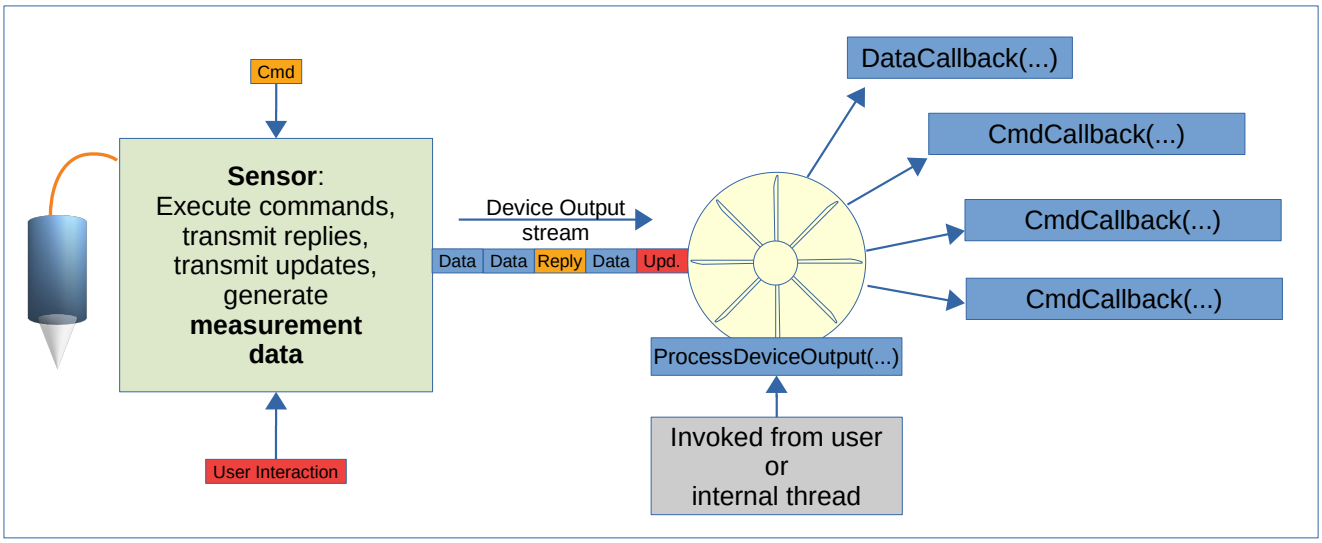


Figure 2.3: Function ProcessDeviceOutput() serves as a message pump. It reads all incoming data, command responses, and updates and delivers their content to the application by calling the registered callback functions.

first, followed by the "general callback function". If there is no special treatment for the command, it is thus not necessary to pass a callback function into every call to ExecCommandAsync().

**Data Callback Function** Similarly to responses, sample data is also delivered through calls to a user-defined callback function (see figure 2.4). This function is registered by calling RegisterSampleDataCallback(). There are two additional parameters given, the maximum number of samples and a timeout in milliseconds. For example, if arguments 1000 and 10 are passed, then this function will be invoked either when 1000 samples have been received or after 10 milliseconds have passed, whichever comes first. Since the device output stream is buffered in a library-internal buffer, even timeouts of zero milliseconds are allowed and are – in certain situations described later – the value of choice (for details on the data callback function see SampleDataCallback).

**Executing Commands** In asynchronous mode, ExecCommandAsync() only queues the device commands to be sent out to the OD7000 device. The call returns almost instantly. In contrast to the synchronous mode, it does not wait for the command to be actually sent and the responses to be received. The transmission of the command will be automatically done by a thread of the library. When later the OD7000 device sends back the responses, they are then processed either by the library itself or through calling ProcessDeviceOutput() (as outlined in figure 2.1). Any incoming responses (and updates) will eventually result in calls to the user-defined callback functions. Any data arriving in the meantime is not disposed as in the synchronous mode but delivered to the user through data callback function calls. Therefore in asynchronous mode, there is no data loss. ExecCommandAsync() can be invoked from a thread other than the thread calling ProcessDeviceOutput().

### 2.2.1 Designing applications in asynchronous mode

A general pattern for writing software that runs in asynchronous mode:

- Open connection
- Register general callbacks
- Register data callback
- *Either* constantly call ProcessDeviceOutput(), e.g. in a GUI timer routine
  *or* StartAutomaticDeviceOutputProcessing() to process OD7000 device output

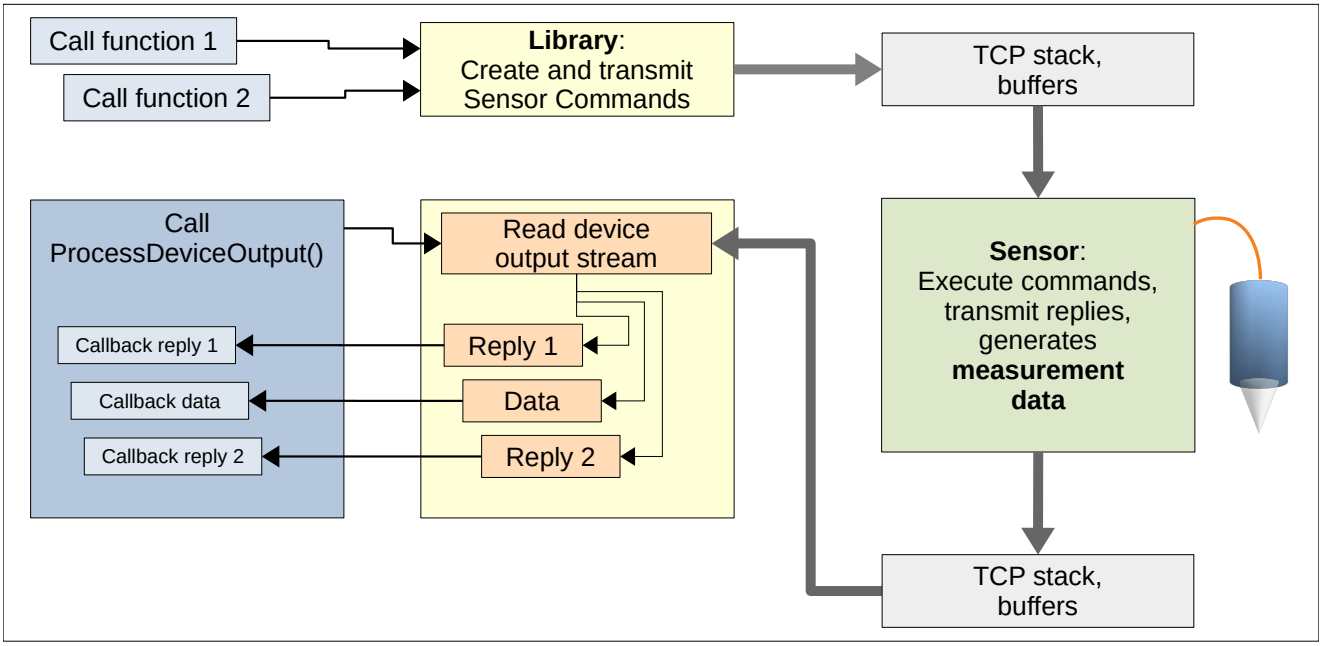


Figure 2.4: Consistent way of receiving data and command responses

**The steps in more detail:**

Upon establishing the connection with OpenConnection(), your application should first register general command callbacks for all device parameters of interest, regardless of whether the command or changed the respective sensor setting has been triggered by your application or somebody else. This implies that commands initially sent by the application itself and any other source of state change are handled uniformly. In this general command callback function you can, for example, put an implementation that updates your application's state (GUI control etc., start/stop data storage or drawing charts, whatever is needed).

After the general command callback function is registered, a data callback function SampleDataCallback needs to be registered with RegisterSampleDataCallback() to process data samples from the device.

If your application calls ProcessDeviceOutput() actively to process device output, e.g. from within a timer routine. Your timer routine should look a bit like this:

Listing 2.1: Calling ProcessDeviceOutput() from a timer routine

```
1  void MyForm::OnTimer()
2  {
3    //return value from ProcessDeviceOutput: negative: error code,
4    //0: timeout in waiting for preset amount of data to be read,
5    //1: successfully read in preset amount of the data,
6    //2: command response has been received,
7    //3: library data sample buffer is not large enough to store preset amount of data.
8    int32_t res = Read_Data_Not_Enough;
9    do
10     res = ProcessDeviceOutput(myConnectionHandle);
11   while (res != Read_Data_Not_Enough);
12 }
```

As shown in listing 2.1, ProcessDeviceOutput() must be called until all pending data and commands are handled.

Note that care must be taken to ensure that all the callback routines does not block for more than maybe a couple of milliseconds. If a callback function is not finished in time, then ProcessDeviceOutput() will not return. Thus the following data samples/commands will not be processed. This causes slow process of device output.

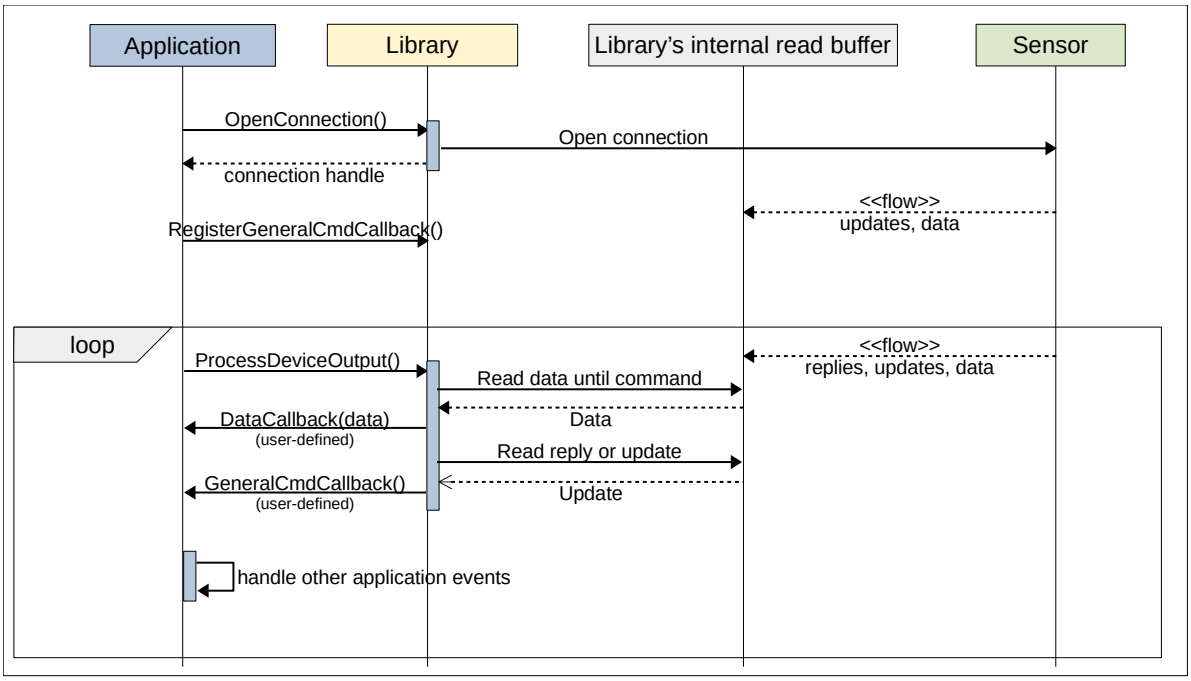


Figure 2.5: Simplified sequence diagram indicating the general steps of an interactive GUI application: Open connection, register callbacks for responses, updates, and data samples, start loop calling ProcessDeviceOutput() which in turn will read the device output stream and invoke the corresponding callback routines of the application

## 2.3 Making use of `_pUser`, the user context pointer

Callback functions are registered either by passing them into a respective RegisterGeneralResponseAndUpdateCallback() function or as an additional argument passed into ExecCommandAsync(). Example:

```
1 Res_t ExecCommandAsync(Conn_h _hConnection, Cmd_h _hCmd, void *_pUser,
      ResponseAndUpdateCallback _pCB, int32_t *_pTicket)
```

One of the arguments, `_pUser`, will be stored and later forwarded to the callback function when the command response has been processed by the device:

```
1 void ResponseCallback(const TRspCallbackInfo _sInfo, Rsp_h _hRsp)
2 {
3 // may use _sInfo.User ....
4 }
```

The library itself does not do anything with `_pUser` apart from storing and forwarding it – which gives the user the opportunity to employ it for application-specific purposes. The code below illustrates a typical C++ pattern, casting `_pUser` to a `this` pointer:

```
1  class MyClass
2  {
3    /*
4     * omitted declarations...:
5     */
6    int MeasuringMode;
7    void CheckState(...);
8
9    void SetMeasuringMethodAsync(int _nMode)
10   {
11     Cmd_h hCmd;
12     NewCommand (CmdID_Measuring_Method, false, &hCmd);
13     AddCommandIntArg (hCmd, _nMode);
14     Check(ExecCommandAsync(hCon, hCmd, this,&MyClass::MeasuringModeCallback, nullptr));
15   }
```

```
16
17   static void MeasuringModeCallback(TCmdCallbackInfo _sInfo, Rsp_h _hRsp)
18   {
19     MyClass * p = static_cast<MyClass *>(_sInfo.User);
20     p->CheckState(_sInfo.State);
21     GetResponseIntArg(_hRsp, 0, &p->MeasuringMode);
22   }
23 }
```

# Chapter 3

# Using Multiple Threads

On a single given connection, calling library functions is generally *not thread-safe*, i.e. they must be called from within one thread or the calls must be serialized/synchronized by the user. This applies to all traditional synchronous commands such as GetNextSamples(), ExecCommand(). All these commands must not be called from different threads without appropriate thread synchronization. The exceptions to this rule are OpenConnection(), OpenSharedConnection() and CloseConnection(). These functions can safely be called from any thread.

In asynchonous mode, ProcessDeviceOutput() must always be invoked from one single thread (or the call must be synchronized, see above) but that thread can – but is not required to – be different from the one the function ExecCommandAsync() is called from.

Note that any callback function, either registered by means of a RegisterGeneralResponseAndUpdateCallback() function or as argument to ExecCommandAsync(), is executed in the context of the ProcessDeviceOutput() call. Consequently, if ProcessDeviceOutput() is called from a different thread, then the callback functions are executed within that thread, too. You should hence bear that in mind when implementing callback functions which may require thread synchronization.

If the connection is *shared*, i.e. a second handle is created based on an existing open “physical” device connection, then these connections can operate on different threads. Details on how to share connections can be found in chapter 4.

The points above can be summarized as follows:

- Calls to library functions are generally *not thread-safe*.
- OpenConnection(), OpenSharedConnection() and CloseConnection() can be called from any thread.
- ProcessDeviceOutput() is not thread-safe but can be called from a thread different from the one calling ExecCommandAsync()
- The above points apply to a single given connection (the same handle). Calls referring to *other* connections do not interfere. Different connections can be handled in different threads.
- The point above can be specifically taken advantage of when a connection is shared (chapter 4).

By default, the library automatically checks the thread ID of the calling function for each connection. This can be turned off, see SetLibConfigFlags() and appendix A.18. All the function calls of one connection in synchronous mode are required to have the same thread ID. In asynchronous mode, the calls to ProcessDeviceOutput() should have the same ID and the calls to ExecCommandAsync() should have the same ID as well. Otherwise the function call returns with error. If because of any reason, functions need to run in different threads but they are properly protected, you can disable the checking of thread ID with SetLibConfigFlags().

# Chapter 4

# Sharing Device Connections

**Motivation** Consider the following situation:

- You are creating an interactive GUI application where users can manipulate certain parameters and view some life data such as the current thickness of a sample.
- For the sake of responsiveness you pick the asynchronous way of sensor communication (see section 2.2), i.e.:
  - You implement different button, combo box, etc. ... handlers that translate user actions to ExecCommandAsync() calls.
  - You implement a general callback function that receives all sensor replies / updates and dispatches all necessary GUI refresh procedures.
  - You call ProcessDeviceOutput() from a timer routine or a dedicated thread.
- But now you also need to offer some long-running functionality such as saving data into a file for a certain period. Or uploading large tables.

**Standard Approaches** You typically handle these requirements with either a state machine-based or a thread-based approach. In the former case you have to process the task in small steps and maintain the book-keeping. In the latter case, you may spawn some task running in parallel that needs to share any command input and reply / data output channels in a (thread-)safe way.

**Shared Connections** Shared connections offer an alternative here:

- Create a thread for the task where you...
- ...open a shared connection
- run the time-consuming task using that connection
- ...and close the connection, wait for the thread to be joined.

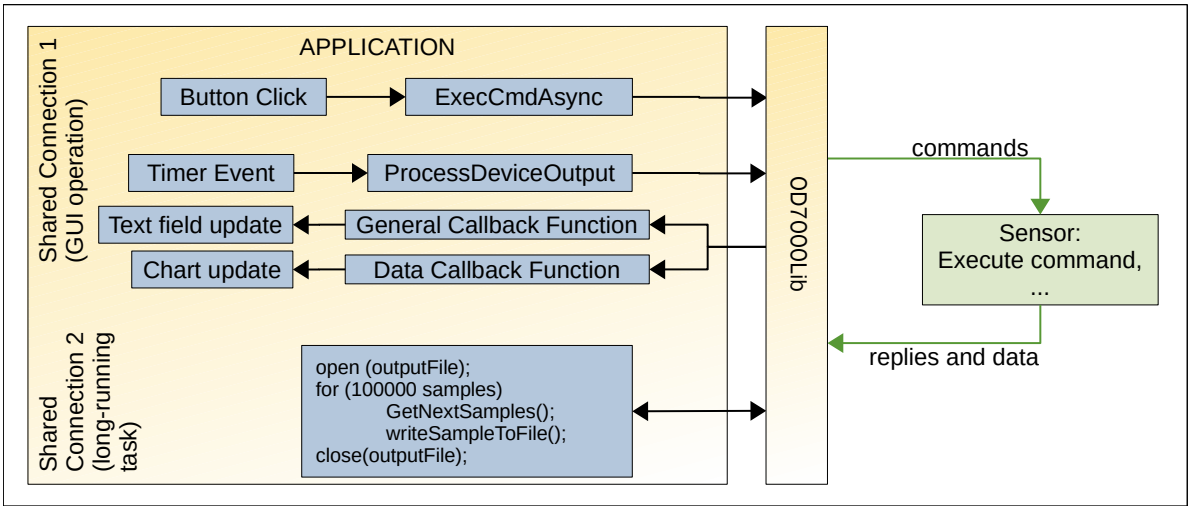


Figure 4.1: Sharing Connections

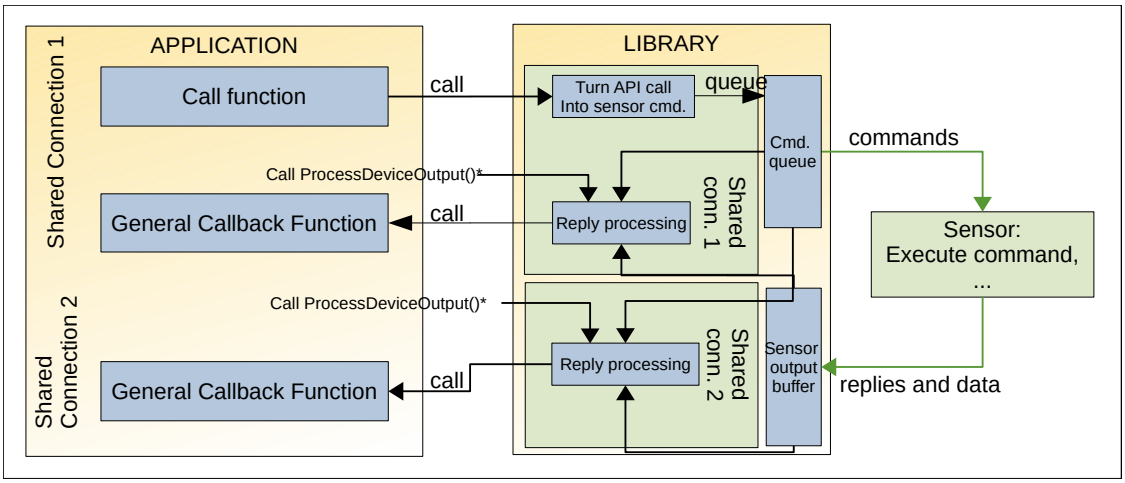


Figure 4.2: Sharing Connections - A closer look.

In the meantime, the rest of the application runs its business as usual. No special sensor communication synchronization required. If the task involves sensor control / state changes, the standard event loop of the application will be notified.

**General Command Response Handler** to enable loose coupling. The reason why this works is that any commands initiated by one shared connection will be normally replied there while – at the same time – a respective *update* will show up in the other shared connection. Your general command response handlers receive both command responses and updates. Figure 4.1 illustrates this.

Figure 4.2 gives a more detailed view of how this is managed inside OD7000Lib: When e. g. ExecCommandAsync() is called, a respective command is stored in a common command queue and then automatically sent to the sensor. Once the sensor has finished t he o peration a nd the reply arrives in the library's sensor output buffer, the content of the reply is associated to it's counterpart in the command queue. These together enable the library to generate the command callback function arguments with the correct "source connection" ID. "Source connection" ID states from which connection the orginal command has been sent. It has the same value of the connection handle. If the "Source connection" ID is 0xFFFFFFFF, value of Invalid_Handle, that means it is an update from OD7000 device, not a reponse of a command from any shared connection.

Shared connections are logical connections to the same device. They communicate with the OD7000 device independently. As stated above, the commands they send will be queued together to be transfered to the device. At the same time they read out and process device output (data and command reponses) separately.

Each shared connection can choose either synchronous or asynchronous mode provided by the library to communicate with the device. Therefore by means of a shared connection, you can even combine synchronous / asynchronous communication. You can keep specific, potentially long-running tasks local in your code, virtually independent of the normal interactive operation.

You can also dedicate shared connections to specific application views such as data charts or life spectra. This may also play nicely with the fact that shared connections – just like connections in general – are thread-protected against each other.

# Part II

# API Reference

# Chapter 7

# Exported functions and data types

## 7.1 Scalar type Cmd_h

**Description**

**Type definition** `typedef Handle_t Cmd_h;`

## 7.2 Scalar type Conn_h

**Description**

**Type definition** `typedef Handle_t Conn_h;`

## 7.3 Scalar type FilePath_t

**Description**

**Type definition** `typedef char const  * FilePath_t;`

## 7.4 Scalar type Handle_t

**Description**

**Type definition** `typedef uint32_t Handle_t;`

## 7.5 Scalar type LibSize_t

**Description**

**Type definition** `typedef int64_t LibSize_t;`

## 7.6 Scalar type Plugin_h

**Description**

**Type definition** `typedef Handle_t Plugin_h;`

## 7.7 Scalar type Rsp_h

**Description**

**Type definition** `typedef Handle_t Rsp_h;`

## 7.8 Structure TErrorInfo

**Description** Information of each single error stored in the buffer returned by LastErrors()

**Type definition**

```
typedef struct {
    Handle_t SourceHandle;
    int32_t ErrorCode;
} TErrorInfo;
```

**Members**

| | |
|---|---|
| SourceHandle | The handle of the connection where the error occurs. |
| ErrorCode | Error code. |

## 7.9 Structure TResponseInfo

**Description** Command response information.

**Type definition**

```
typedef struct {
    uint32_t CmdID;
    int32_t Ticket;
    uint32_t Flag;
    uint32_t ParamCount;
} TResponseInfo;
```

**Members**

| | |
|---|---|
| CmdID | Command/response ID. |
| Ticket | Internal ticket for the original command. |
| Flag | Response flag see A.4. |
| ParamCount | Number of parameters in the response. |

## 7.10 Structure TRspCallbackInfo

**Description** General command response information in callback function.

**Type definition**

```
typedef struct {
    void * User;
    int32_t * State;
    int32_t Ticket;
    Handle_t SourceConnection;
} TRspCallbackInfo;
```

**Members**

| | |
|---|---|
| User | Value as passed to RegisterGeneralResponseAndUpdateCallback or ExecCommandAsync() function. Forwarded but not interpreted by the library. See also section 2.3 of the documentation. |
| State | Status of command response: negative: error or warning code, `0`: no error. If `*_State` is set to `1` by the callback function, general command callback function for this response will not be called. |
| Ticket | Internal ticket for command. |
| SourceConnection | Handle of the connection/plugin which has originally issued the command for current response. If response has been sent out by device as an update, `0xFFFFFFFF` will be given out. |

**More details** As stated in 2.2, command callback function can be passed to the library either by ExecCommandAsync() or RegisterGeneralResponseAndUpdateCallback(). The callback function passed through ExecCommandAsync() is only for that one specific command. This callback function will be called before the general command callback function, which is passed through RegisterGeneralResponseAndUpdateCallback(). If in the callback function of ExecCommandAsync(), "*_State" is set to 1 by the user, general command callback function for this response will not be called, as normally. Note the State pointer, `_State`, will always be non-NULL.

## 7.11 Structure TSampleSignalGeneralInfo

**Description** Structure to represent the general information of output signals.

**Type definition**

```
typedef struct {
    uint32_t InfoIndex;
    int32_t PeakSignalCount;
    int32_t GlobalSignalCount;
    int32_t ChannelCount;
} TSampleSignalGeneralInfo;
```

**Members**

| | |
|---|---|
| InfoIndex | Index of sample signal data format; increases with changing of the data format. |
| PeakSignalCount | Number of peak signals in data sample. For single channel device, all the signals are regarded as global signal. |
| GlobalSignalCount | Global signal count in data sample. |
| ChannelCount | Device channel count. |

## 7.12 Structure TSampleSignalInfo

**Description** Structure to represent the detailed information of each output signal.

**Type definition**

```
typedef struct {
    uint16_t SignalID;
    int16_t DataType;
} TSampleSignalInfo;
```

**Members**

| | |
|---|---|
| SignalID | Signal ID. |
| DataType | Data type. For a detailed definition see section A.3. |

## 7.13 Callback ResponseAndUpdateCallback

**Description** The format of callback function upon receiving command response.

**Prototype**

```
typedef void(* ResponseAndUpdateCallback)(
    TRspCallbackInfo _sInfo,
    Rsp_h _hRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _sInfo | Struct contains the general information about the response (see **TRspCallbackInfo**) |
| in | _hRsp | Response handle. |

## 7.14 Callback SampleDataCallback

**Description** Data sample callback function, which is called to pass device data samples.

**Prototype**

```
typedef void(* SampleDataCallback)(
    void * _pUser,
    int32_t _nState,
    int64_t _nSampleCount,
    const double * _pSampleBuffer,
    LibSize_t _nSizePerSample,
    TSampleSignalGeneralInfo _sSignalGeneralInfo,
    TSampleSignalInfo * _pSignalInfo
);
```

**Arguments**

| | | |
|---|---|---|
| in | _pUser | Value as passed to RegisterSampleDataCallback. Forwarded but not interpreted by the library. See also section 2.3 of the reference manual. |
| in | _nState | Data reading status: negative: error or warning code, 0: time out in waiting for preset amount of data to be read, 1: successfully read in preset amount of the data, 2: command response has been received, 3: library data Sample buffer is not large enough to store preset amount of data. |
| in | _nSampleCount | Number of data samples received. |
| in | _pSampleBuffer | Buffer which stores received data sample. Buffer is maintained by library. |
| in | _nSizePerSample | Size of each data sample. |
| in | _sSignalGeneralInfo | General information of signals. |
| in | _pSignalInfo | Array which contains the detailed description of each signal. |

## 7.15 Function ActivateAutoBufferMode()

**Description** Automatically save data from device to the target buffer.

**Prototype**

```
Res_t ActivateAutoBufferMode(
    Conn_h _hConnection,
    double * _pBuffer,
    int64_t _nSampleCount,
    LibSize_t * _pBufferSizeInBytes
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _pBuffer | A double pointer to the buffer where the data is written to. |
| in | _nSampleCount | Number of the sample to be written to the buffer. |
| inout | _pBufferSizeInBytes | Size of the target buffer in byte. If the target buffer is not large enough or _pBuffer is not assigned, required minimum size is written to *_pBufferSizeInBytes. |
| return | | Zero on success. Negative error or warning code. |

**More details**

Normally the library will not automatically read the data from the device. It only temporarily saves the data to an internal circular buffer. When the user calls GetNextSamples(), sample data will be read in and processed. However the user can also set the library to automatically read in a specific number of the data and ask the library to save the data to a buffer *("auto buffer save mode")* by using the above function.

When this function is called, the library will begin to save the next `_nSampleCount` of data into the buffer from the position where the data was last read. It will not start with latest data, i.e. it will not automatically jump to the last valid position of the internal buffer. If you want to only start to save the new data, call the function FlushConnectionBuffer() first.

All the saved data has the same format as from the function GetNextSamples(), i.e. the data is saved sample-wise not signal- wise. If signal distance and intensity of peak 1 has been ordered from the device, the data saved in the buffer will be like:

*sample 1: distance, intensity;*
*sample 2: distance, intensity;*
*sample 3: distance, intensity;*
*...;*
*sample n: distance, intensity*

If enough number of data has been saved, the library will quit this automatic saving mode. If for any reason, the output signals of the device have been changed and the signals required by the user are not among the device output signals any more, the library will also quit this mode.

If the size`_pBufferSizeInBytes` passed over is not large enough, the library will not save any data to the buffer and will return with error. It will also give back at the parameter `_pBufferSizeInBytes` the miminum size of the buffer required for saving `_nSampleCount` of the sample. The user could use this function twice: first time to acquire the buffer size needed for his application; second time to save the data to an appropriate buffer.

This function is only valid for synchronous connection.

## 7.16 Function AddCommandBlobArg()

**Description**

Add binray data argument to the command.

**Prototype**

```
Res_t AddCommandBlobArg(
    Cmd_h _hCmd,
    const int8_t * _pArg,
    int32_t _nLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _pArg | A pointer, pointing to where binary data resides |
| in | _nLength | Size of the binary data |
| return | | Zero on success. Negative error or warning code. |

## 7.17 Function AddCommandFloatArg()

**Description** Add float argument to the command.

**Prototype**

```
Res_t AddCommandFloatArg(
    Cmd_h _hCmd,
    float _nArg
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _nArg | Float argument. |
| return | | Zero on success. Negative error or warning code.. |

## 7.18 Function AddCommandFloatArrayArg()

**Description** Add float array argument to the command.

**Prototype**

```
Res_t AddCommandFloatArrayArg(
    Cmd_h _hCmd,
    const float * _pArg,
    int32_t _nLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _pArg | Pointer to the float array argument. |
| in | _nLength | Length of the float array. |
| return | | Zero on success. Negative error or warning code. |

## 7.19 Function AddCommandIntArg()

**Description** Add integer argument to the command.

**Prototype**

```
Res_t AddCommandIntArg(
    Cmd_h _hCmd,
    int32_t _nArg
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _nArg | Integer argument. |
| return | | Zero on success. Negative error or warning code.. |

## 7.20 Function AddCommandIntArrayArg()

**Description** Add integer array argument to the command.

**Prototype**

```
Res_t AddCommandIntArrayArg(
    Cmd_h _hCmd,
    const int32_t * _pArg,
    int32_t _nLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _pArg | Pointer to the integer array argument. |
| in | _nLength | Length of the integer array. |
| return | | Zero on success. Negative error or warning code. |

## 7.21 Function AddCommandStringArg()

**Description** Add string argument to the command.

**Prototype**

```
Res_t AddCommandStringArg(
    Cmd_h _hCmd,
    const char * _pArg,
    int32_t _nLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _pArg | A zero-terminated argument string. |
| in | _nLength | Length of the string. |
| return | | Zero on success. Negative error or warning code. |

## 7.22 Function CancelCHRDeviceAutoSearch()

**Description** Cancel automatic OD7000 device search in unblocking search mode.

**Prototype**

```
void CancelCHRDeviceAutoSearch();
```

## 7.23 Function ClearErrors()

**Description** Clear errors.

**Prototype**

```
Res_t ClearErrors(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | When the valid connection handle is given, the error of that specific connection is cleared. When Invalid_Handle is passed, the errors of all the connections are cleared. |
| return | | Zero on success, negative in case of an error or warning. |

## 7.24 Function CloseConnection()

**Description** Close connection to OD7000 device.

**Prototype**

```
Res_t CloseConnection(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative error or warning code.. |

## 7.25 Function CmdNameToID()

**Description** Translate device command name to Lib command ID.

**Prototype**

```
int32_t CmdNameToID(
    const char * _strCmd,
    int32_t _nLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _strCmd | Device command name in zero-terminated string. |
| in | _nLength | Length of the command name. |
| return | | Corresponding command ID |

## 7.26 Function DeactivateAutoBufferMode()

**Description** Manually quit automatic buffer data save mode.

**Prototype**

```
Res_t DeactivateAutoBufferMode(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative value: error or warning code. |

## 7.27 Function DestroyHandle()

**Description** Release object related to the handle.

**Prototype**

```
void DestroyHandle(
    Handle_t _Handle
);
```

**Arguments**

| | | |
|---|---|---|
| in | _Handle | Handle which to be released |

**More details** The command handle will be automatically invalidated after ExecCommand() or ExecCommandAsync()

## 7.28 Function DetectedCHRDeviceInfo()

**Description** Get connection information of automatically detected OD7000 devices.

**Prototype**

```
Res_t DetectedCHRDeviceInfo(
    char * _aDeviceInfos,
    LibSize_t * _pnSize
);
```

**Arguments**

| | | |
|---|---|---|
| in | _aDeviceInfos | Pointer to the buffer where the connection information string of detected OD7000 should be saved to (zero-termination is included). Information for different devices is separated by ";". |
| inout | _pnSize | Size of the given string buffer. If it is not large enough or _aDeviceInfos is not assigned, the minimum required size is written back (including zero termination). |
| return | | Zero on success. Negative error or warning code. |

**More details** The returned connection information string for each detected OD7000 device is similar to the connection string passed over in OpenConnection(). The only difference is that it also contains the device type. The information string for single device can be directly passed to OpenConnection(), with "CHR_Unspecified" as device type. Information of different devices is separated by ";".

## 7.29 Function ErrorCodeToString()

**Description** Get error string from error code.

**Prototype**

```
Res_t ErrorCodeToString(
    int32_t _nErrorCode,
    char * _aErrorStr,
    LibSize_t * _pnSize
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nErrorCode | Error code. |
| in | _aErrorStr | Pointer to the string buffer where the zero-terminated error string should be saved to (zero termination included). |
| inout | _pnSize | Size of the given string buffer. The minimum required size is written back (including zero termination). |
| return | | Zero on success, negative in case of an error or warning. |

## 7.30 Function ExecCommand()

**Description** Send command to device/plugin and wait for the response, i.e. synchronous command sending.

**Prototype**

```
Res_t ExecCommand(
    Handle_t _hHandle,
    Cmd_h _hCmd,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hHandle | Connection handle (returned by OpenConnection() or OpenSharedConnection()) or plugin handle (returned by AddConnectionPlugin()). |
| in | _hCmd | Command handle returned by NewCommand(). |
| out | _phRsp | Pointer to a variable receiving the response handle. |
| return | | Zero on success. Negative error or warning code. |

**More details** This function sends the command and waits for the response. All the samples, if any, received in between will be ignored/skipped. The command is passed over based on the handle returned by NewCommand(). The response is also given by a form of handle. User can query related information of the response with functions like GetResponseInfo(), GetResponseArgType(), GetResponseIntArg()... This function is only valid for synchronous connection.

## 7.31 Function ExecCommandAsync()

**Description** Send command to devicee/plugin without waiting for the response, i.e. asynchronous command sending.

**Prototype**

```
Res_t ExecCommandAsync(
    Handle_t _hHandle,
    Cmd_h _hCmd,
    void * _pUser,
    ResponseAndUpdateCallback _pCB,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hHandle | Connection handle (returned by OpenConnection() or OpenSharedConnection()) or plugin handle (returned by AddConnectionPlugin()). |
| in | _hCmd | Command handle returned by NewCommand(). |
| in | _pUser | Can be freely used to forward user information to the function _pCB points to. Passed to _pCB but not interpreted by the library. See also section 2.3 of the reference manual. |
| in | _pCB | Callback function for current command when response from device comes back. |
| out | _pTicket | Pointer to a variable receiving the internal ticket for the issued device command. |
| return | | Zero on success. Negative error or warning code. |

**More details** This function queues the command, without waiting for it to be actually sent and the response to be received. The reponse will be passed over to the user through ResponseAndUpdateCallback(), which is either given here in `_pCB` for the current command or in the general command callback function registered with RegisterGeneralResponseAndUpdateCallback(). The command is based on the handle returned by NewCommand(). This function is only valid for asynchronous connection.

## 7.32 Function FlushConnectionBuffer()

**Description** Flush all the received data and command response, move the read pointer to the last valid position in the data stream from the device.

**Prototype**

```
Res_t FlushConnectionBuffer(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative error or warning code. |

**More details** This function can be used to flush away all the old received data. Or when user software is out of sync with the library (may caused by waiting too long to read from the library), i.e. the position of the user reader is not valid any more, to move user reader to the last valid position. This function is only valid for synchronous connection.

## 7.33 Function GetAutoBufferSavedSampleCount()

**Description** Get the number of the sample already saved into the buffer in the automatic buffer data save mode.

**Prototype**

```
Res_t GetAutoBufferSavedSampleCount(
    Conn_h _hConnection,
    int64_t * _pnSampleCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnSampleCount | Pointer to a variable receiving the number of saved samples. |
| return | | Zero on success. Negative value: error or warning code. |

## 7.34 Function GetAutoBufferStatus()

**Description** Get the status of automatic buffer data save.

**Prototype**

```
int32_t GetAutoBufferStatus(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | negative: error or warning code; otherwise the status of automatic buffer data save, see Appendix A.9 |

## 7.35 Function GetConnectionMode()

**Description** Get device connection mode.

**Prototype**

```
int32_t GetConnectionMode(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | negative error or warning code; otherwise see Appendix A.7 |

## 7.36 Function GetConnectionOutputSignalInfos()

**Description** Get the signal information of the data sample output by the library.

**Prototype**

```
Res_t GetConnectionOutputSignalInfos(
    Conn_h _hConnection,
    TSampleSignalGeneralInfo * _pSignalGeneralInfo,
    TSampleSignalInfo * _pSignalInfoBuf,
    LibSize_t * _pnSignalInfoBufSizeInBytes
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pSignalGeneralInfo | A pointer to the struct, where the general information of signals in data sample (see **TSampleSignalGeneralInfo**) is returned. |
| in | _pSignalInfoBuf | Pointer to the buffer where detailed description of each signal should be saved to. |
| inout | _pnSignalInfoBufSizeInBytes | Size of the given buffer. The minimum required size is written back. |
| return | | Zero on success. Negative error or warning code.. |

**More details** The returned signal information is valid for output data through all data acquisition functions: GetNextSamples(), GetLastSample(), ActivateAutoBufferMode() and SampleDataCallback(). By default library outputs every signal data in double format, user can change the signal data type to orginal data type output by the device through function SetOutputDataFormatMode().

## 7.37 Function GetCurrentAvailableSampleCount()

**Description** Number of current available sample to be read.

**Prototype**

```
Res_t GetCurrentAvailableSampleCount(
    Conn_h _hConnection,
    int64_t * _pnSampleCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnSampleCount | Current available sample count. |
| return | | Zero on success. Negative error or warning code.. |

## 7.38 Function GetDeviceConnectionInfo()

**Description** Get device connection information.

**Prototype**

```
Res_t GetDeviceConnectionInfo(
    Conn_h _hConnection,
    char * _aConnInfoStr,
    LibSize_t * _pnSize
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _aConnInfoStr | Pointer to the string buffer where the connection information string should be saved to (zero termination included). |
| inout | _pnSize | Size of the given string buffer. The minimum required size is written back (including zero termination). |
| return | | Zero on success. Negative error or warning code.. |

## 7.39 Function GetDeviceOutputSignals()

**Description** Get IDs of the signals output by the device.

**Prototype**

```
Res_t GetDeviceOutputSignals(
    Conn_h _hConnection,
    int32_t * _aSignalIDs,
    int32_t * _pSignalCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _aSignalIDs | Pointer to the integer buffer where ID of device output signals should be save to. |
| inout | _pSignalCount | In: the Size of the given integer buffer, out: the number of signals output by the device |
| return | | Zero on success. Negative: error or warning code. |

**More details** The returned signals are the original signals output by the OD7000 They should be at least the signals ordered by all the shared connections to this device.

## 7.40 Function GetDeviceType()

**Description** Get device type.

**Prototype**

```
int32_t GetDeviceType(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | negative error or warning code; otherwise see Appendix A.6 |

## 7.41 Function GetHandleType()

**Description** Get handle type.

**Prototype**

```
int32_t GetHandleType(
    Handle_t _Handle
);
```

**Arguments**

| | | |
|---|---|---|
| in | _Handle | Handle whose type to be checked |
| return | | handle type A.2.. |

## 7.42 Function GetLastSample()

**Description** Get the last valid sample from device.

**Prototype**

```
Res_t GetLastSample(
    Conn_h _hConnection,
    const double ** _ppData,
    LibSize_t * _pnSizePerSample,
    TSampleSignalGeneralInfo * _pSignalGeneralInfo,
    TSampleSignalInfo ** _pSignalInfo
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _ppData | A pointer to the double sample buffer where data is saved. |
| out | _pnSizePerSample | A pointer to an integer variable which stores the size of each data sample. |
| out | _pSignalGeneralInfo | A pointer to the struct, which contains the general information of signals in data sample (see **TSampleSignalGeneralInfo**). |
| out | _pSignalInfo | A pointer to the array, where detailed description of each signal is stored one by one. |
| return | | 1: sample is available; 0: no sample has been received from device; negative: error or warning code. |

**More details** This function is used to get the last/newest received valid sample from the device. The sample saved in the returned buffer has the same format as in GetNextSamples(). If no new sample has been received since the last function call of “GetLastSample”, the same sample will be returned. By calling this function, user read pointer will also jump to the last valid position in the internal circular buffer, which is used to temporarily store the output from the device. This means all the samples received in between will be ignored/skipped.

Like GetNextSamples(), this function is only valid for synchronous connection. It also cannot be called when the library automatically saves data sample (in "AutoBufferSave" mode).

## 7.43 Function GetNextSamples()

**Description** Get the next available samples from the device.

**Prototype**

```
Res_t GetNextSamples(
    Conn_h _hConnection,
    int64_t * _pSampleCount,
    const double ** _ppData,
    LibSize_t * _pnSizePerSample,
    TSampleSignalGeneralInfo * _pSignalGeneralInfo,
    TSampleSignalInfo ** _pSignalInfo
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| inout | _pSampleCount | User passes over the number of samples to be read, lib gives back the actual number of read samples. |
| out | _ppData | A pointer to the double sample buffer where data is saved. |
| out | _pnSizePerSample | A pointer to an integer variable which stores the size of each data sample in byte. |
| out | _pSignalGeneralInfo | A pointer to the struct, which contains the general information of signals in data sample (see **TSampleSignalGeneralInfo**). |
| out | _pSignalInfo | A pointer to the array, where detailed description of each signal is stored one by one. |
| return | | negative: error or warning code, 0 : present available data is fewer than the required number, 1 : successfully read in required number of the data, 2 : command response has been received, 3 : library data sample buffer is not large enough to store preset amount of data (A.10). |

**More details**

If there a are samples available from the device, the function will fill a memory l ovation into pData where the sample data can be picked up by the user. The bu˙er where the data is stored is maintained by the library, and the user is told where it is located. The user specifies in SampleCount how many samples he intends to read, SampleCount returns with actual available number.
All the returned samples are saved one after another in the returned buyer.
Signal values are generally given in double precision floating point format and have been rescaled. Signal values are generally given in double precision floating point format and are rescaled if necessary. For certain signals, such as sample counter or encoder counter, it is important to maintain their full resolution for the whole value range. By using double precision floating point as consistent output format for all signals, this resolution is guaranteed and thus counters given in double format can safely be converted back to integers.
If the original data format from the OD7000 instead of double, is prefered, SetOutputData-FormatMode() can be used to change. User can check with _pSignalGeneralInfo and _pSignalInfo about the details of delivered samples signals and their data format.
If, for any reason, device output signals have been changed and the signals ordered by the user are not among the output signals of the device anymore, this function will return an error.
Since the library will continuously save incoming data from the device into an internal bu˙er, if the user has not read from this internal bu˙er, i.e. get sample or send command, for a long time, the user will be out of sync with the data bu˙er. In this case, this function can fail. The user can go back to sync with data buffer by jumping to the last valid position (start position of the last unfinished telegram) in the bu˙er by calling function FlushConnectionBuffer().
This function is only valid for synchronous connection. It also cannot be called when the library automatically saves data sample (in "AutoBufferSave" mode).

## 7.44 Function GetOutputDataFormatMode()

**Description** Get the mode of library output data format.

**Prototype**

```
Res_t GetOutputDataFormatMode(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | negative: error or warning code; 0 - double; 1 - raw. |

## 7.45 Function GetResponseArgType()

**Description** Get response argument type.

**Prototype**

```
Res_t GetResponseArgType(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    int32_t * _pArgType
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArgType | A pointer to the integer variable receiving the argument type. |
| return | | Zero on success. Negative error or warning code. |

## 7.46 Function GetResponseBlobArg()

**Description** Get binary response argument.

**Prototype**

```
Res_t GetResponseBlobArg(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    const int8_t ** _pArg,
    int32_t * _pLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArg | A pointer to the buffer where the binary data is restored. The buffer is maintained by the library. |
| out | _pLength | A pointer to the integer variable, where the size of the binary data. |
| return | | Zero on success. Negative error or warning code. |

## 7.47 Function GetResponseFloatArg()

**Description** Get float response argument.

**Prototype**

```
Res_t GetResponseFloatArg(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    float * _pArg
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArg | A pointer to the float variable receiving the argument value. |
| return | | Zero on success. Negative error or warning code. |

## 7.48 Function GetResponseFloatArrayArg()

**Description** Get float array response argument.

**Prototype**

```
Res_t GetResponseFloatArrayArg(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    const float ** _pArg,
    int32_t * _pLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArg | A pointer to the array, where the float array argument is returned. Array is maintained by the library. |
| out | _pLength | A pointer to the integer variable receiving the length of the float array argument. |
| return | | Zero on success. Negative error or warning code. |

## 7.49 Function GetResponseInfo()

**Description** Get response information.

**Prototype**

```
Res_t GetResponseInfo(
    Rsp_h _hRsp,
    TResponseInfo * _pRspInfo
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| out | _pRspInfo | A pointer to the struct, where the response information (see **TResponseInfo**) is returned. |
| return | | Zero on success. Negative error or warning code. |

**More details** This function is to get general response information from the response handle, which is either returned by ExecCommand() or given by ResponseAndUpdateCallback().

## 7.50 Function GetResponseIntArg()

**Description** Get integer response argument.

**Prototype**

```
Res_t GetResponseIntArg(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    int32_t * _pArg
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArg | A pointer to the integer variable receiving the argument value. |
| return | | Zero on success. Negative error or warning code. |

## 7.51 Function GetResponseIntArrayArg()

**Description** Get integer array response argument.

**Prototype**

```
Res_t GetResponseIntArrayArg(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    const int32_t ** _pArg,
    int32_t * _pLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArg | A pointer to the array, where the integer array argument is returned. Array is maintained by the library. |
| out | _pLength | A pointer to the integer variable receiving the length of the integer array argument. |
| return | | Zero on success. Negative error or warning code. |

## 7.52 Function GetResponseStringArg()

**Description** Get string response argument.

**Prototype**

```
Res_t GetResponseStringArg(
    Rsp_h _hRsp,
    uint32_t _nIndex,
    const char ** _pArg,
    int32_t * _pLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nIndex | Argument index, start with 0 |
| out | _pArg | A pointer to the zero-terminated string containing the argument value |
| out | _pLength | A pointer to the integer variable, where the length of the string - including the terminating zero - is returned. |
| return | | Zero on success. Negative error or warning code. |

## 7.53 Function GetSingleOutputSampleSize()

**Description** Get the size of each data sample output by the library.

**Prototype**

```
Res_t GetSingleOutputSampleSize(
    Conn_h _hConnection,
    LibSize_t * _pnSampleSize
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnSampleSize | A pointer to an integer variable which stores the size of each data sample. |
| return | | Zero on success. Negative error or warning code.. |

**More details** The sample size information is valid for output data through all data acquisition functions: GetNextSamples(), GetLastSample(), ActivateAutoBufferMode() and SampleDataCallback().

## 7.54 Function IsCHRDeviceAutoSearchFinished()

**Description** Query whether OD7000 device search in unblocking mode has been finished.

**Prototype**

```
Res_t IsCHRDeviceAutoSearchFinished();
```

## 7.55 Function LastErrors()

**Description** Get the information of the last errors (maximum 1024 for one connection)

**Prototype**

```
Res_t LastErrors(
    Conn_h _hConnection,
    TErrorInfo * _aErrorInfos,
    LibSize_t * _pnBufSizeInBytes
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | When the valid connection handle is given, function returns the errors of that specific connection. When Invalid_Handle is passed, the errors of all the connections will be returned. |
| in | _aErrorInfos | Pointer to the buffer where error information should be save to (in the form of an array of `TErrorInfo` (alignment 4)). |
| inout | _pnBufSizeInBytes | Size of the given error buffer in bytes. The minimum required size is written back. |
| return | | number of errors, or negative in case of an error or warning. |

## 7.56 Function NewCommand()

**Description** Create new command.

**Prototype**

```
Res_t NewCommand(
    uint32_t _nCmdID,
    int32_t _bQuery,
    Cmd_h * _hCmd
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nCmdID | Command ID. |
| in | _bQuery | Whether command is query or not. |
| out | _hCmd | A pointer to a variable receiving the command handle. |
| return | | Zero on success. Negative error or warning code.. |

**More details** This function creates a new commmand object inside library and returns the sociated command handle. Definition of each OD7000 device command, except OD7000Lib internal com-mands (please refer to appendix D), is the same as the corresponding device command defined in the device manual. Based on the returned command handle, user can add command related parameters, if any, with functions like AddCommandIntArg(). Afterwards, user passes this handle in ExecCommand() and ExecCommandAsync() as the reference to the command to be sent. The created command handle can also be used in and PerformLibOperation() to perform OD7000Lib internal operation. Upon executing the command, the command object will be automatically deleted from the library, the related command handle should be not further used.

## 7.57 Function NewCommandFromString()

**Description** Translate string command to internal command and give back related command handle.

**Prototype**

```
int32_t NewCommandFromString(
    const char * _strCommand,
    Cmd_h * _hCmd
);
```

**Arguments**

| | | |
|---|---|---|
| in | _strCommand | Zero terminated string command. |
| out | _hCmd | A pointer to an unsigned integer variable receiving the command handle. |
| return | | Zero on success. Negative error or warning code. |

**More details** With this function, the command ID and the parameters are already extracted from the given string. The returned command handle can be directly used in ExecCommand() and ExecCommandAsync().

## 7.58 Function OpenConnection()

**Description** Connect to OD7000 device.

**Prototype**

```
Res_t OpenConnection(
    const char * _strConnectionInfo,
    int32_t _nDeviceType,
    int32_t _eConnectionMode,
    LibSize_t _nDevBufSizeInBytes,
    Conn_h * _pHandle
);
```

**Arguments**

| | | |
|---|---|---|
| in | _strConnectionInfo | A zero-terminated string that contains the connection information. |
| in | _nDeviceType | For a list of device types see A.6. |
| in | _eConnectionMode | Connection mode: synchornous or asynchronous A.7. |
| in | _nDevBufSizeInBytes | The size of library internal buffer for storing device output (only powers of 2 allowed (e.g. 16x1024x1024)). By default (when _nDevBufSizeInBytes<=0 ), 32MB. |
| out | _pHandle | A pointer to a variable receiving the connection handle. This handle is passed when calling other functions related to this connection. |
| return | | Zero on success. Negative error or warning code.. |

**More details** For OD7000, the library uses packet communication protocol. In this case, if any port number is passed to OpenConnection() function, it should always be the port number for packet protocol (default port **7891**).

The connection information string for serial connection looks like `COM15, Baud: 921600, Handshake: On`. If baudrate is not specified, the default value **921600** will be applied. If handshake is not specified, it will be regarded as **On**. For ethernet connection the string looks like `192.168.170.2, Port: 7891`. If port number is not specified, the default value **7891** will be taken.

If the connection information string is directly returned by library function DetectedCHRDeviceInfo(), user can pass over type "CHR_Unspecified" (A.6.), since the device type information is already contained in the string. Otherwise suitable device type has to be specified.

## 7.59 Function OpenSharedConnection()

**Description** Open an extra connection shared with the exising one.

**Prototype**

```
Res_t OpenSharedConnection(
    Conn_h _nExistingConnection,
    int32_t _eConnectionMode,
    Conn_h * _pHandle
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nExistingConnection | Handle of existing connection, which to be shared with. |
| in | _eConnectionMode | Connection mode: synchornous or asynchronous A.7. |
| out | _pHandle | A pointer to a variable receiving the connection handle. This handle is passed when calling other functions related to this connection. |
| return | | Zero on success. Negative error or warning code.. |

**More details** All the shared connections have only one physical connection to the same device. Shared connections read data and send command independently to the device. However since they actually connect to the same device, commands, which are sent, of course, affect all the connections. Shared connections can be put into different threads. Closing one does not affect other ones. When all the shared connections are closed down, the physical connection to the device will be automatically closed.

## 7.60 Function PerformLibOperation()

**Description** Perform 0D7000Lib internal operations.

**Prototype**

```
Res_t PerformLibOperation(
    Conn_h _hConnection,
    Cmd_h _hCmd,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _hCmd | Command handle returned by NewCommand(). |
| out | _phRsp | Pointer to a variable receiving the response handle. |
| return | | Zero on success. Negative error or warning code. |

## 7.61 Function ProcessDeviceOutput()

**Description** Process device output. All the data samples and command responses are passed through callback functions.

**Prototype**

```
Res_t ProcessDeviceOutput(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | negative: error or warning code, 0: timeout in waiting for preset amount of data to be read, 1: successfully read in preset amount of the data, 2: command response has been received, 3: library data sample buffer is not large enough to store preset amount of data (A.10). |

**More details** This function returns under following conditions:

- Time out at waiting for data samples
- Enough data samples have been received
- Command response has been received
- Library data sample buffer has been filled up
- Error

For related settings of the sample number and timeout please refer to RegisterSampleDataCallback(). When this function is called, received data samples and command responses are passed through the registered callback functions ResponseAndUpdateCallback() and SampleDataCallback(). This function is only valid for asynchronous connection. It can also not to be called when the library automatically processes device output (see StartAutomaticDeviceOutputProcessing()).

## 7.62 Function RegisterGeneralResponseAndUpdateCallback()

**Description** Register callback function for all the commands.

**Prototype**

```
Res_t RegisterGeneralResponseAndUpdateCallback(
    Conn_h _hConnection,
    void * _pUser,
    ResponseAndUpdateCallback _pGenCBFct
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _pUser | Can be freely used to forward user information to the function _pCB points to. Passed to _pCB() but not interpreted by the library. See also section 2.3 of the reference manual. |
| in | _pGenCBFct | Callback function for all the responses/updates (see ResponseAndUpdateCallback). |
| return | | Zero on success. Negative error or warning code. |

**More details** Register general callback function for the responses of all the commands and updates from the device in asynchronous connection. Through ExecCommandAsync(), a specific callback function can also be passed over, howevever only for that specific command. When the response comes back from the 0D7000 the specific callback function, if any, will be called first and then this general callback function. The updates, which are automatically sent by the device, can only be delivered to the user through this general callback function.

## 7.63 Function RegisterSampleDataCallback()

**Description** Register data sample callback function and set data processing related settings.

**Prototype**

```
Res_t RegisterSampleDataCallback(
    Conn_h _hConnection,
    int64_t _nMaxSampleCount,
    int32_t _nReadSampleTimeOut,
    void * _pUser,
    SampleDataCallback _pOnReadSample
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nMaxSampleCount | Maximum number of samples to be read during single device output processing. |
| in | _nReadSampleTimeOut | Timeout in millisecond for waiting for _nReadSampleCount of samples. If _nReadSampleTimeOut is set to 0, only available samples are read without waiting. |
| in | _pUser | Can be used to carry user information (see 2.3). It is not interpreted by the library. When _pOnReadSample is called, this value will be passed into it. |
| in | _pOnReadSample | Sample callback function to pass over samples (format see SampleDataCallback), which is called either by function ProcessDeviceOutput() or by library automatic output processing. |
| return | | Zero on success. Negative error or warning code.. |

## 7.64 Function ResponseToString()

**Description** Output response in string format.

**Prototype**

```
Res_t ResponseToString(
    Rsp_h _hRsp,
    char * _aResponseStr,
    LibSize_t * _pnSize
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _aResponseStr | Pointer to the buffer where the response string should be saved to (zero-termination is included). |
| inout | _pnSize | Size of the given string buffer. If it is not large enough or _aResponseStr is not assigned, the minimum required size is written back (including zero termination). |
| return | | Zero on success. Negative error or warning code. |

**More details** Based on the given response handle, this function outputs the response in string format: $Cmd-Name Argument 1; Argument 2;... The string will be written to the provided char buffer. If the buffer is not large enough (based on `_pnSize`), minimum required size will be returned back in `_pnSize`.

## 7.65 Function SetLibConfigFlags()

**Description** Set library configuration flags.

**Prototype**

```
void SetLibConfigFlags(
    uint32_t _nFlag
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nFlag | CHRLib_Flag_Rsp_Deactivate_Auto_Buffer: set library to automatically quit auto buffer save mode when response/update has been received from the device, by default this flag is not set.<br>CHRLib_Flag_Auto_Change_Data_Buffer_Size: set library to automatically resize data sample buffer to store preset number of data samples in RegisterSampleDataCallback or GetNextSamples(), by default this flag is set.<br>CHRLib_Flag_Check_Thread_ID: set library to check the thread ID of function call. When the connection is in synchronous mode, all the function calls related to command executing and data reading should be from the same thread. When the connection is in asynchronous mode, all the function calls to ExecCommandAsync() should be in the same thread, all the function calls to ProcessDeviceOutput() should also be in one thread. By default this flag is set. |

## 7.66 Function SetLibLogFileDirectory()

**Description** Set log file directory property.

**Prototype**

```
Res_t SetLibLogFileDirectory(
    FilePath_t _pstrDirectory,
    LibSize_t _nMaxFileSizeInKiB,
    int32_t _nMaxFileNumber
);
```

**Arguments**

| | | |
|---|---|---|
| in | _pstrDirectory | Log file directory. |
| in | _nMaxFileSizeInKiB | Maximum size of one log file in KiB. |
| in | _nMaxFileNumber | Maximum number of log files in the directory. If in the log directory, the number of log files are larger than _nMaxFileNumber, the oldest log file(s) will be automatically deleted. |
| return | | Zero on success. Negative: error or warning code. |

## 7.67 Function SetLibLogLevel()

**Description** Set the level for logging library messages. The message which has the higher or equal level will be logged.

**Prototype**

```
Res_t SetLibLogLevel(
    int32_t _nLevel
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nLevel | Minimum logging level. By default, the logging level is 2. |
| return | | Zero on success. Non-zero otherwise. |

## 7.68 Function SetOutputDataFormatMode()

**Description** Set the mode of library output data format **(Use carefully!)**.

**Prototype**

```
Res_t SetOutputDataFormatMode(
    Conn_h _hConnection,
    int32_t _eMode
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _eMode | Output data format mode: 0 - double; 1 - raw. |
| return | | Zero on success. Negative error or warning code.. |

**More details** By default, all the data samples stored in the data buffer, either given by GetNextSamples()/ GetLastSample() or through SampleDataCallback() or stored in the user buffer in "AutoBuffer-Save" mode, are in double format. With the same format, it is easy for user to read out the data. However it may unnecessarily occupy a large storage space. To save storage space, user can set the library to output the raw data from the device. In this mode, raw sample data for each signal is saved/packed directly behind each other in the buffer. In order to read out the data correctly, user needs to know however the exact format of each signal data.

## 7.69 Function SetSampleDataBufferSize()

**Description** Set the size of buffer to store data sample.

**Prototype**

```
Res_t SetSampleDataBufferSize(
    Conn_h _hConnection,
    LibSize_t _nBufferSizeInBytes
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nBufferSizeInBytes | Size of data sample buffer. |
| return | | Zero on success. Negative error or warning code.. |

**More details** Data sample buffer is passed to the user at data sample callback function SampleDataCallback() or GetNextSamples()/ GetLastSample() function. It is managed by the library. The default size of the buffer is 512KB. By default, library automatically increases its size, based on the ordered sample number from GetNextSamples() and RegisterSampleDataCallback(). However user can set the sample buffer to have a fixed size (in case of uncontrolled memory increasement) with SetLibConfigFlags() and set the size with the above funcion.

## 7.70 Function StartAutomaticDeviceOutputProcessing()

**Description** Start library automatic processing of device output.

**Prototype**

```
Res_t StartAutomaticDeviceOutputProcessing(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative error or warning code.. |

**More details** Upon calling this function, the library will start a new thread to automatically process device ouput. In this mode, user himself cannot use ProcessDeviceOutput() to trigger output processing any more. All the command response and data sample are passed to the user through callback functions in the form of ResponseAndUpdateCallback() and SampleDataCallback() respectively. The callback functions can be registered through function RegisterGeneralResponseAndUpdate-Callback() and RegisterSampleDataCallback(). This function is only valid for asynchronous connection.

## 7.71 Function StartCHRDeviceAutoSearch()

**Description** Start automatic OD7000 device search.

**Prototype**

```
Res_t StartCHRDeviceAutoSearch(
    int32_t _nConnectionType,
    int32_t _bSBlockingSearch
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nConnectionType | 0: search both serial and TCPIP connection; 1: search only serial connection; 2: search only TCPIP connection for OD7000 connection (see A.18). |
| in | _bSBlockingSearch | Whether start search function return immediately (unblocking search) or wait until search is finished (blocking search). |
| return | | Zero on success. Negative: error or warning code. |

**More details** Only one device search can be started at one time. The detected device information can be retrieved through DetectedCHRDeviceInfo(). IsCHRDeviceAutoSearchFinished() is used to check the status of the unblocking device search.

## 7.72 Function StartConnectionDataStream()

**Description** Start data stream for one specific connection.

**Prototype**

```
Res_t StartConnectionDataStream(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative error or warning code.. |

**More details** This function tells the library to send data samples again through SampleDataCallback() to user/client or enables reading data samples through GetNextSamples() and GetLastSample() for **current** connection. It does not affect the data status of the OD7000 device.

## 7.73 Function StopAutomaticDeviceOutputProcessing()

**Description** Stop library automatic processing of device output.

**Prototype**

```
Res_t StopAutomaticDeviceOutputProcessing(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative error or warning code.. |

**More details** This function ends the library thread to automatically process device output. After it, user is allowed to call ProcessDeviceOutput() again. This function is also only valid for asynchronous connection.

## 7.74 Function StopConnectionDataStream()

**Description** Stop data stream for one specific connection.

**Prototype**

```
Res_t StopConnectionDataStream(
    Conn_h _hConnection
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| return | | Zero on success. Negative error or warning code.. |

**More details** This function only stops the library from sending data for the **current** connection. When connection is under synchronous mode, GetNextSamples(), GetLastSample(), ActivateAutoBufferMode() returns error. When connection is under asynchronous mode, data will be not delivered through SampleDataCallback().It does not, however, affect the data status of the 0D7000 device.

## 7.75 Function UploadConfocalCalibrationTableFromFile()

**Description** Synchronously upload confocal calibration table from file to device.

**Prototype**

```
Res_t UploadConfocalCalibrationTableFromFile(
    Conn_h _hConnection,
    FilePath_t _strFullFileName,
    uint32_t _nProbeSerialNumber,
    uint32_t _nCHRTableIndex
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _strFullFileName | Calibration table file name (full name including file path). |
| in | _nProbeSerialNumber | Sensor probe serial number. |
| in | _nCHRTableIndex | Index of calibration table stored in device (from 0 to 15). |
| return | | Zero on success. Negative: error or warning code. |

**More details** This function can only be used in synchronous connection.

## 7.76 Function UploadFirmware()

**Description** Synchronously upload firmware file to device.

**Prototype**

```
Res_t UploadFirmware(
    Conn_h _hConnection,
    FilePath_t _strFullFileName
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _strFullFileName | Firmware file name (full name including file path). |
| return | | Zero on success. Negative: error or warning code. |

**More details** Uploading firmware to OD7000 devices may take up to a few minutes based on different device types. After uploading, the device will be restarted automatically with the new firmware and the library will reconnect to the device. This function can only be used in synchronous connection.

## 7.77 Function UploadRefractiveIndexTableFromFile()

**Description** Synchronously upload refractive index table from file to device.

**Prototype**

```
Res_t UploadRefractiveIndexTableFromFile(
    Conn_h _hConnection,
    FilePath_t _strFullFileName,
    const char * _strTableName,
    uint32_t _nCHRTableIndex,
    float _nRefSRI
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _strFullFileName | Refractive index table file name (full name including file path). |
| in | _strTableName | Table name. |
| in | _nCHRTableIndex | Table index stored in device (from 1 to 16). |
| in | _nRefSRI | Reference refractive index (see below). |
| return | | Zero on success. Negative: error or warning code. |

**More details** This function can only be used in synchronous connection.

## 7.78 Function UploadWaveLengthTableFromFile()

**Description** Synchronously upload wavelength table from file to device.

**Prototype**

```
Res_t UploadWaveLengthTableFromFile(
    Conn_h _hConnection,
    FilePath_t _strFullFileName
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _strFullFileName | Wavelength table file name (full name including file path). |
| return | | Zero on success. Negative: error or warning code. |

**More details** This function can only be used in synchronous connection.

# Appendix A

# Predefined Constants

## A.1 Result Codes

Please look at the `OD7000LibErrorDef.h` header for more details.

```
#define RES_LEVEL_SUCCESS 0
#define RES_LEVEL_INFORMATION 1
#define RES_LEVEL_WARNING 2
#define RES_LEVEL_ERROR 3
#define CHR_SUCCESS ((res) >= 0)
#define CHR_INFORMATION \
    ((((uint32_t)res) >> 30) == RES_LEVEL_INFORMATION)
#define CHR_WARNING ((((uint32_t)res) >> 30) == RES_LEVEL_WARNING)
#define CHR_ERROR ((((uint32_t)res) >> 30) == RES_LEVEL_ERROR)
#define SCS_SUCCESS ((int32_t)0x00000000L)
#define WRN_CONNECTION_ATTEMPT_UNDERWAY ((int32_t)0xA0041400L)
#define WRN_AUTOFLUSH_OCCURRED ((int32_t)0xA0047600L)
#define ERR_ALLOCATE ((int32_t)0xE0000C00L)
#define ERR_FREE ((int32_t)0xE0000D00L)
#define ERR_REALLOCATE ((int32_t)0xE0000E00L)
#define ERR_WRONG_HANDLE_TYPE ((int32_t)0xE0001E00L)
#define ERR_INVALID_HANDLE ((int32_t)0xE0001F00L)
#define ERR_NULLPTR ((int32_t)0xE0002000L)
#define ERR_BUFSIZE_TOO_SMALL ((int32_t)0xE0002100L)
#define ERR_UNCAUGHT ((int32_t)0xE0006300L)
#define ERR_BUFSIZE_POWER_OF_TWO ((int32_t)0xE0008600L)
#define ERR_UNKNOWN ((int32_t)0xE000FF00L)
#define ERR_OPEN_FILE ((int32_t)0xE0010A00L)
#define ERR_FILE_ALREADY_EXISTS ((int32_t)0xE0011400L)
#define ERR_NO_SUCH_FILE ((int32_t)0xE0011500L)
#define ERR_OPEN_COMPORT ((int32_t)0xE0020A00L)
#define ERR_COMPORT_WRITE_TIMEOUT ((int32_t)0xE0022800L)
#define ERR_READ_COMPORT ((int32_t)0xE0023200L)
#define ERR_WRITE_COMPORT ((int32_t)0xE0023300L)
#define ERR_GET_STATE_COMPORT ((int32_t)0xE0023400L)
#define ERR_SET_STATE_COMPORT ((int32_t)0xE0023500L)
#define ERR_FLUSH_SERIAL_CONNECTION_STREAM ((int32_t)0xE0026500L)
#define ERR_CONNECT_SOCKET ((int32_t)0xE0030A00L)
#define ERR_ALLOCATE_SOCKET ((int32_t)0xE0030C00L)
#define ERR_READ_CLOSED_SOCKET ((int32_t)0xE0031600L)
#define ERR_SOCKET_TIMEOUT ((int32_t)0xE0032800L)
#define ERR_READ_SOCKET ((int32_t)0xE0033200L)
#define ERR_WRITE_SOCKET ((int32_t)0xE0033300L)
#define ERR_GET_STATE_SOCKET ((int32_t)0xE0033400L)
#define ERR_SET_STATE_SOCKET ((int32_t)0xE0033500L)
#define ERR_SELECT_SOCKET ((int32_t)0xE0036600L)
#define ERR_CONNECTION_NOT_OPEN ((int32_t)0xE0041500L)
#define ERR_WRONG_CONNECTION_MODE ((int32_t)0xE0041E00L)
```

```
#define ERR_UNSUPPORTED_DEVICE_FUNCTIONALITY ((int32_t)0xE0042200L)
#define ERR_RESPONSE_TIMEOUT ((int32_t)0xE0042900L)
#define ERR_COMMAND_SEND_TIMEOUT ((int32_t)0xE0042A00L)
#define ERR_RESYNC_FAILED ((int32_t)0xE0046900L)
#define ERR_SIGNAL_MISSING ((int32_t)0xE0046A00L)
#define ERR_CONNECTION_OUT_OF_SYNC ((int32_t)0xE0046D00L)
#define ERR_AUTOFLUSH_FAILED ((int32_t)0xE0047700L)
#define ERR_AUTO_PROCESS_ACTIVE ((int32_t)0xE0048400L)
#define ERR_AUTO_SAVE_ACTIVE ((int32_t)0xE0048500L)
#define ERR_CONNECTION_INVALID_PARAMETER ((int32_t)0xE0048700L)
#define ERR_CONNECTION_UNKNOWN_DEVTYPE ((int32_t)0xE0048800L)
#define ERR_DATASTREAM_STOPPED ((int32_t)0xE0048900L)
#define ERR_NO_SIGNALS_SELECTED ((int32_t)0xE0048A00L)
#define ERR_DEVSEARCH_ALLOCATE ((int32_t)0xE0050C00L)
#define ERR_DEVSEARCH_REALLOCATE ((int32_t)0xE0050E00L)
#define ERR_DEVSEARCH_UNDERWAY ((int32_t)0xE0051400L)
#define ERR_DEVSEARCH_CANCELED ((int32_t)0xE0052B00L)
#define ERR_DEVSEARCH_GETIFADDRS ((int32_t)0xE0056E00L)
#define ERR_DEVSEARCH_UPNP_ALLOCATE_SEND_SOCKET ((int32_t)0xE0056F00L)
#define ERR_DEVSEARCH_UPNP_ALLOCATE_RECV_SOCKET ((int32_t)0xE0057000L)
#define ERR_DEVSEARCH_UPNP_SET_IFADDR ((int32_t)0xE0057100L)
#define ERR_DEVSEARCH_UPNP_BIND_MULTICAST ((int32_t)0xE0057200L)
#define ERR_DEVSEARCH_UPNP_REUSE_PORT ((int32_t)0xE0057300L)
#define ERR_DEVSEARCH_UPNP_ADD_MEMBERSHIP ((int32_t)0xE0057400L)
#define ERR_DEVSEARCH_UPNP_SEARCH ((int32_t)0xE0057500L)
#define ERR_DEVSEARCH_UNKNOWN ((int32_t)0xE005FF00L)
#define ERR_UNSUPPORTED_FUNCTIONALITY ((int32_t)0xE0062200L)
#define ERR_CLIENT_READER_ERROR ((int32_t)0xE0066700L)
#define ERR_INTERNAL_READER_ERROR ((int32_t)0xE0066800L)
#define ERR_DATAFMT_MISSING ((int32_t)0xE0066B00L)
#define ERR_DATAFMT_INVALID ((int32_t)0xE0066C00L)
#define ERR_LIB_STREAM_OUT_OF_SYNC ((int32_t)0xE0066D00L)
#define ERR_UNSUPPORTED_FUNCTIONALITY_DEBUG ((int32_t)0xE0067D00L)
#define ERR_WRONG_THREAD ((int32_t)0xE0067E00L)
#define ERR_SPLINE_CALC_FAILED ((int32_t)0xE0068000L)
#define ERR_WSASTARTUP ((int32_t)0xE0076400L)
#define ERR_TONUMBER ((int32_t)0xE0083C00L)
#define ERR_TOSTRING ((int32_t)0xE0083D00L)
#define ERR_CMD_RESPONSE ((int32_t)0xE0097800L)
#define ERR_CMD_RESPONSE_PARAM_INDEX ((int32_t)0xE0097A00L)
#define ERR_CMD_RESPONSE_PARAM_TYPE ((int32_t)0xE0097B00L)
#define ERR_CMD_RESPONSE_ID_DIFFERENT ((int32_t)0xE0097C00L)
#define ERR_FIRMWARE_UPLOAD ((int32_t)0xE0097F00L)
#define ERR_CMDBUF_EMPTY ((int32_t)0xE0098100L)
#define ERR_CMDBUF_INVALID_READPOS ((int32_t)0xE0098200L)
#define ERR_CMDBUF_INVALID_TICKET ((int32_t)0xE0098300L)
#define ERR_CMD_INVALID_PARAM ((int32_t)0xE0098B00L)
#define ERR_CMD_INVALID_FILE ((int32_t)0xE0098C00L)
#define ERR_CMD_UNKNOWN ((int32_t)0xE009FF00L)
```

## A.2 Constants: Handle Types

```
const Handle_t Invalid_Handle = ((Handle_t)-1);
const int32_t Handle_Unknown_Type = -1;
const int32_t Handle_Connection = 0;
const int32_t Handle_Command = 1;
const int32_t Handle_Response = 2;
const int32_t Handle_Plugin = 3;
```

## A.3 Constants: Sample Data Types

```
const int16_t Data_Type_Unsigned_Char = 0;
const int16_t Data_Type_Signed_Char = 1;
const int16_t Data_Type_Unsigned_Short = 2;
const int16_t Data_Type_Signed_Short = 3;
const int16_t Data_Type_Unsigned_Int32 = 4;
const int16_t Data_Type_Signed_Int32 = 5;
const int16_t Data_Type_Float = 6;
const int16_t Data_Type_Double = 255;
```

## A.4 Constants: Response flag

```
const uint32_t Rsp_Flag_Query = 0x0001;
const uint32_t Rsp_Flag_Error = 0x8000;
const uint32_t Rsp_Flag_Warning = 0x4000;
const uint32_t Rsp_Flag_Update = 0x2000;
```

## A.5 Constants: Device Response Parameter Types

```
const int32_t Rsp_Param_Type_Integer = 0;
const int32_t Rsp_Param_Type_Float = 1;
const int32_t Rsp_Param_Type_String = 2;
const int32_t Rsp_Param_Type_Byte_Array = 4;
const int32_t Rsp_Param_Type_Integer_Array = 254;
const int32_t Rsp_Param_Type_Float_Array = 255;
```

## A.6 Constants: Connection Constants

```
const int32_t CHR_2_Device = 1;
```

## A.7 Constants: Connection Mode

```
const int32_t Connection_Synchronous = 0;
const int32_t Connection_Asynchronous = 1;
```

## A.8 Constants: Library Data Formats

```
const int32_t Output_Data_Format_Double = 0;
const int32_t Output_Data_Format_Raw = 1;
```

## A.9 Constants: Automatic Buffer Saving Constants

```
const int32_t Auto_Buffer_Error = -1;
const int32_t Auto_Buffer_Saving = 0;
const int32_t Auto_Buffer_Finished = 1;
const int32_t Auto_Buffer_Received_Response = 2;
const int32_t Auto_Buffer_Deactivated = 3;
const int32_t Auto_Buffer_UnInit = 4;
```

## A.10 Constants: Return value of data read

```
const int32_t Read_Data_Not_Enough = 0;
const int32_t Read_Data_Success = 1;
const int32_t Read_Data_Response = 2;
const int32_t Read_Data_Buffer_Small = 3;
```

## A.11 Constants: Device Command Specifier ID Constants

```
const uint32_t CmdID_Output_Signals = 0x58444f53;
const uint32_t CmdID_Firmware_Version = 0x00524556;
const uint32_t CmdID_Measuring_Method = 0x00444d4d;
const uint32_t CmdID_Full_Scale = 0x00414353;
const uint32_t CmdID_Scan_Rate = 0x005a4853;
const uint32_t CmdID_Data_Average = 0x00445641;
const uint32_t CmdID_Spectrum_Average = 0x00535641;
const uint32_t CmdID_Serial_Data_Average = 0x53445641;
const uint32_t CmdID_Refractive_Indices = 0x00495253;
const uint32_t CmdID_Abbe_Numbers = 0x00454241;
const uint32_t CmdID_Refractive_Index_Tables = 0x00545253;
const uint32_t CmdID_Lamp_Intensity = 0x0049414c;
const uint32_t CmdID_Optical_Probe = 0x004e4553;
const uint32_t CmdID_Confocal_Detection_Threshold = 0x00524854;
const uint32_t CmdID_Peak_Separation_Min = 0x004D4350;
const uint32_t CmdID_Duty_Cycle = 0x00594344;
const uint32_t CmdID_Detection_Window_Active = 0x00414d4c;
const uint32_t CmdID_Detection_Window = 0x00445744;
const uint32_t CmdID_Number_Of_Peaks = 0x00504f4e;
const uint32_t CmdID_Peak_Ordering = 0x00444f50;
const uint32_t CmdID_Dark_Reference = 0x004b5244;
const uint32_t CmdID_Continuous_Dark_Reference = 0x4b445243;
const uint32_t CmdID_Start_Data_Stream = 0x00415453;
const uint32_t CmdID_Stop_Data_Stream = 0x004f5453;
const uint32_t CmdID_Light_Source_Auto_Adapt = 0x004c4141;
const uint32_t CmdID_CCD_Range = 0x00415243;
const uint32_t CmdID_Median = 0x5844454d;
const uint32_t CmdID_Analog_Output = 0x58414e41;
const uint32_t CmdID_Encoder_Counter = 0x53504525;
const uint32_t CmdID_Encoder_Counter_Source = 0x53434525;
const uint32_t CmdID_Encoder_Preload_Function = 0x46504525;
const uint32_t CmdID_Encoder_Trigger_Enabled = 0x45544525;
const uint32_t CmdID_Encoder_Trigger_Property = 0x50544525;
const uint32_t CmdID_Device_Trigger_Mode = 0x4d525425;
const uint32_t CmdID_Download_Spectrum = 0x444c4e44;
const uint32_t CmdID_Save_Settings = 0x00555353;
const uint32_t CmdID_Download_Upload_Table = 0x4C424154;
```

## A.12 Constants: Spectra Types

```
const int32_t Spectrum_Raw = 0;
const int32_t Spectrum_Confocal = 1;
const int32_t Spectrum_FT = 2;
const int32_t Spectrum_2D_Image = 3;
```

## A.13 Constants: Table Types

```
const int32_t Table_Confocal_Calibration = 1;
const int32_t Table_WaveLength = 2;
const int32_t Table_Refractive_Index = 3;
const int32_t Table_Dark_Correction = 4;
```

## A.14 Constants: Device Trigger Modes

```
const int32_t Device_Trigger_Mode_Free_Run = 0;
const int32_t Device_Trigger_Mode_Wait_Trigger = 1;
const int32_t Device_Trigger_Mode_Trigger_Each = 2;
const int32_t Device_Trigger_Mode_Trigger_Window = 3;
```

## A.15 Constants: Encoder Counter Input Source

```
const int32_t Encoder_Counter_Trigger_Source_A0 = 0;
const int32_t Encoder_Counter_Trigger_Source_B0 = 1;
const int32_t Encoder_Counter_Trigger_Source_A1 = 2;
const int32_t Encoder_Counter_Trigger_Source_B1 = 3;
const int32_t Encoder_Counter_Trigger_Source_A2 = 4;
const int32_t Encoder_Counter_Trigger_Source_B2 = 5;
const int32_t Encoder_Counter_Trigger_Source_A3 = 6;
const int32_t Encoder_Counter_Trigger_Source_B3 = 7;
const int32_t Encoder_Counter_Trigger_Source_A4 = 8;
const int32_t Encoder_Counter_Trigger_Source_B4 = 9;
const int32_t Encoder_Counter_Trigger_Source_SyncIn = 10;
const int32_t Encoder_Counter_Source_Quadrature = 15; const
int32_t Encode_Trigger_Source_Immediate = 15;
```

## A.16 Constants: Encoder Preload Event Property Constants

```
const int32_t Encoder_Preload_Config_Once = 0;
const int32_t Encoder_Preload_Config_Eachtime = 1;
const int32_t Encoder_Preload_Config_Trigger_RisingEdge = 0;
const int32_t Encoder_Preload_Config_Trigger_FallingEdge = 2;
const int32_t Encoder_Preload_Config_Trigger_OnEdge = 0;
const int32_t Encoder_Preload_Config_Trigger_OnLevel = 4;
const int32_t Encoder_Preload_Config_Active = 8;
```

## A.17 Constants: Library Configuration Flags

```
const uint32_t CHRLib_Flag_Rsp_Deactivate_Auto_Buffer = 1;
const uint32_t CHRLib_Flag_Auto_Change_Data_Buffer_Size = 2;
const uint32_t CHRLib_Flag_Check_Thread_ID = 4;
```

## A.18 Constants: Device Search Constants

```
const int32_t Search_Both_Serial_TCPIP_Connection = 0;
const int32_t Search_Only_Serial_Connection = 1;
const int32_t Search_Only_TCPIP_Connection = 2;
```

## A.19 Constants: Performance save setting

```
const uint32_t OPID_Save_Performance = 0x5356504D;
const int32_t Performance_Snapshot_Mode = 0;
const int32_t Performance_Complete_Mode = 1;
```

# Appendix B

# The OD7000Lib API grouped by topic

## B.1 Asynchronous Mode

Functions, structures, constants and other entities related to using the asynchronous mode.
Also see

- Automatic Device Output Processing
- Callback Functions

Functions, structures and constants related to this

- ExecCommandAsync()
- ProcessDeviceOutput()

## B.2 Automatic Buffer Saving

Functions, structures, constants and other entities related to using the automatic buffer saving feature.
Also see

- Automatic Buffer Saving Constants

Functions, structures and constants related to this

- ActivateAutoBufferMode()
- DeactivateAutoBufferMode()
- GetAutoBufferSavedSampleCount()
- GetAutoBufferStatus()

## B.3 Automatic Device Output Processing

Functions, structures and constants related to this

- StartAutomaticDeviceOutputProcessing()
- StopAutomaticDeviceOutputProcessing()

## B.4 Automatic Device Search

Functions for automatic device search.

Also see

- Device Search Constants

Functions, structures and constants related to this

- CancelCHRDeviceAutoSearch()
- DetectedCHRDeviceInfo()
- IsCHRDeviceAutoSearchFinished()
- StartCHRDeviceAutoSearch()

## B.5 Callback Functions

Everything related to callback functions.

Also see

- DataCallback Function
- Update and Response Callback Function

## B.6 Command Response

Functions that relate to get the information from responses

Also see

- Device Response Parameter Types
- Response flag
- Update and Response Callback Function

Functions, structures and constants related to this

- CheckCmdID()
- GetResponseArgType()
- GetResponseBlobArg()
- GetResponseFloatArg()
- GetResponseFloatArrayArg()
- GetResponseInfo()
- GetResponseIntArg()
- GetResponseIntArrayArg()
- GetResponseStringArg()

- ReadSingleFloatArrayParamRsp()
- ReadSingleFloatParamRsp()
- ReadSingleIntArrayParamRsp()
- ReadSingleIntParamRsp()
- ResponseToString()
- **`TResponseInfo`**

## B.7 Connection

Functionality related to establishing a connection, retrieving information about it and closing it.

Also see

- Connection Constants
- Connection Information
- Connection Mode

Functions, structures and constants related to this

- CloseConnection()
- OpenConnection()
- OpenSharedConnection()

## B.8 Connection Information

Functions, structures and constants related to this

- GetConnectionMode()
- GetDeviceConnectionInfo()
- GetDeviceType()

## B.9 DataCallback Function

Functions, structures and constants related to this

- RegisterSampleDataCallback
- SampleDataCallback

## B.10 Device Command

Functions that create commands for OD7000Lib, OD7000 or plugin Also see

- Device Command Specifier ID Constants
- Device Trigger Modes
- Spectra Types
- Table or Firmware Upload
- Table Types
- Update and Response Callback Function

Functions, structures and constants related to this

- AddCommandBlobArg()
- AddCommandFloatArg()
- AddCommandFloatArrayArg()
- AddCommandIntArg()
- AddCommandIntArrayArg()
- AddCommandStringArg()
- CmdIDToName()
- CmdNameToID()
- ExecNoParamCmd()
- ExecSingleFloatArrayParamCmd()
- ExecSingleFloatParamCmd()
- ExecSingleIntArrayParamCmd()
- ExecSingleIntParamCmd()
- NewCommand()
- NewCommandFromString()

## B.11 Device Information

Functions that to get the device information
Functions, structures and constants related to this

- GetDeviceChannelCount()
- GetDeviceOutputSignals()

## B.12 Error Handling

Functions, structures, constants and other entities related to error information
Also see

- Result Code Constants

Functions, structures and constants related to this

- ClearErrors()
- ErrorCodeToString()
- LastErrors()
- Res_t
- **`TErrorInfo`**

## B.13 Handles

Functions, structures, constants and other entities related to all kinds of handle
Also see

- Handle Types

Functions, structures and constants related to this

- DestroyHandle()
- GetHandleType()

## B.14 Library Configuration

Functions to set OD7000Lib configuration
Also see

- Library Configuration Flags
- Logging Configuration

Functions, structures and constants related to this

- SetLibConfigFlags()
- SetSampleDataBufferSize()

## B.15 Plugins

Functions to configure plugins
Functions, structures and constants related to this

- AddConnectionPlugIn()

## B.16 Sample Data

Functions, structures, constants and other entities related to retrieve data.
Also see

- Automatic Buffer Saving
- DataCallback Function
- Library Data Formats
- Sample Data Types

Functions, structures and constants related to this

- GetConnectionOutputSignalInfos()
- GetCurrentAvailableSampleCount()
- GetOutputDataFormatMode()
- GetSingleOutputSampleSize()
- SetOutputDataFormatMode()
- StartConnectionDataStream()
- StopConnectionDataStream()
- **`TSampleSignalGeneralInfo`**
- **`TSampleSignalInfo`**

## B.17 Synchronous Mode

Functions, structures, constants and other entities related to using the synchronous mode.
Also see

- Automatic Buffer Saving
- Table or Firmware Upload

Functions, structures and constants related to this

- ExecCommand()
- FlushConnectionBuffer()
- GetLastSample()
- GetNextSamples()

## B.18 Update and Response Callback Function

Functions, structures and constants related to this

- RegisterGeneralResponseAndUpdateCallback
- ResponseAndUpdateCallback
- **`TRspCallbackInfo`**

# Appendix C

# C/C++ Convenience and helper functions

## C.1 Function CheckCmdID()

**Description** Validate the command ID originally passed from inside a response.

**Prototype**

```
Res_t CheckCmdID(
    Rsp_h _hRsp,
    uint32_t _nCmdID
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nCmdID | Command ID. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.2 Function CmdIDToName()

**Description** Convert a numeric command ID to its mnemonic (short string) form.

**Prototype**

```
void CmdIDToName(
    uint32_t _nCmdID,
    char * _strCmd
);
```

**Arguments**

| | | |
|---|---|---|
| in | _nCmdID | Command ID. |
| out | _strCmd | A buffer of at least 4 bytes size. The mnemonic form of the device command is returned in this buffer. This function will not take care of zero- terminating the buffer, so if you pass a buffer bigger than 4 bytes make sure to zero-terminate it yourself. |

## C.3 Function DarkReference()

**Description** Take dark reference, wait for the response and return lowest scan rate.

**Prototype**

```
Res_t DarkReference(
    Conn_h _hConnection,
    float * _pMinFrequency,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pMinFrequency | (Virtual) scan rate in Hertz at which the CCD would be saturated by the stray light, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.4 Function DownloadConfocalCalibrationTable()

**Description** Download calibration table from device with one measurement channel.

**Prototype**

```
Res_t DownloadConfocalCalibrationTable(
    Conn_h _hConnection,
    uint32_t _nCHRTableIndex,
    const int8_t ** _pTable,
    int32_t * _pnTableLength,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nCHRTableIndex | Index of calibration table (0-16). |
| out | _pTable | A pointer to a byte pointer where the calibration table is stored (managed by the library), only used for synchronous connection. |
| out | _pnTableLength | A pointer to an integer which stores the length of the calibration table (in bytes), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.5 Function DownloadDeviceSpectrum()

**Description** Download spectrum from device with one measuring channel.

**Prototype**

```
Res_t DownloadDeviceSpectrum(
    Conn_h _hConnection,
    int32_t _nSpecType,
    const int16_t ** _pSpectrumBuffer,
    int32_t * _pSpectrumLength,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nSpecType | Spectrum type (definition see Appendix A.12). |
| out | _pSpectrumBuffer | A pointer to a short/word pointer where the spectrum is stored, only used for synchronous connection. |
| out | _pSpectrumLength | A pointer to an integer which stores the length of the spectrum, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.6 Function DownloadRefractiveIndexTable()

**Description** Download refractive index table from device.

**Prototype**

```
Res_t DownloadRefractiveIndexTable(
    Conn_h _hConnection,
    int32_t _nTableIndex,
    const int8_t ** _pTable,
    int32_t * _pnTableLength,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nTableIndex | Index of refractive table (1-17). |
| out | _pTable | A pointer to a byte pointer where the calibration table is stored (managed by the library), only used for synchronous connection. |
| out | _pnTableLength | A pointer to an integer which stores the length of the wavelength table (in bytes), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.7 Function DownloadWaveLengthTable()

**Description** Download wavelength table from device.

**Prototype**

```
Res_t DownloadWaveLengthTable(
    Conn_h _hConnection,
    const int8_t ** _pTable,
    int32_t * _pnTableLength,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pTable | A pointer to a byte pointer where the calibration table is stored (managed by the library), only used for synchronous connection. |
| out | _pnTableLength | A pointer to an integer which stores the length of the wavelength table (in bytes), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.8 Function ExecNoParamCmd()

**Description** Execute a device command without parameters.

**Prototype**

```
Res_t ExecNoParamCmd(
    Conn_h _hConnection,
    uint32_t _nCmdID,
    int32_t _bQuery,
    int32_t * _pTicket,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nCmdID | Command ID. |
| in | _bQuery | Whether command is query or not. |
| out | _pTicket | Pointer to a variable receiving the internal ticket for the issued device command. |
| out | _phRsp | Pointer to a buffer receiving the response handle. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.9 Function ExecSingleFloatArrayParamCmd()

**Description** Execute a device command with one float array parameter.

**Prototype**

```
Res_t ExecSingleFloatArrayParamCmd(
    Conn_h _hConnection,
    uint32_t _nCmdID,
    int32_t _bQuery,
    const float * _pParam,
    int32_t _nParamlength,
    int32_t * _pTicket,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nCmdID | Command ID. |
| in | _bQuery | Whether command is query or not. |
| in | _pParam | The float array parameter to the device command. |
| in | _nParamlength | The length of the float array in elements. |
| out | _pTicket | Pointer to a variable receiving the internal ticket for the issued device command. |
| out | _phRsp | Pointer to a buffer receiving the response handle. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.10 Function ExecSingleFloatParamCmd()

**Description** Execute a device command with one float parameter.

**Prototype**

```
Res_t ExecSingleFloatParamCmd(
    Conn_h _hConnection,
    uint32_t _nCmdID,
    int32_t _bQuery,
    float _nParam,
    int32_t * _pTicket,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nCmdID | Command ID. |
| in | _bQuery | Whether command is query or not. |
| in | _nParam | The float parameter to the device command. |
| out | _pTicket | Pointer to a variable receiving the internal ticket for the issued device command. |
| out | _phRsp | Pointer to a buffer receiving the response handle. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.11 Function ExecSingleIntArrayParamCmd()

**Description** Execute a device command with one integer array parameter.

**Prototype**

```
Res_t ExecSingleIntArrayParamCmd(
    Conn_h _hConnection,
    uint32_t _nCmdID,
    int32_t _bQuery,
    const int32_t * _pParam,
    int32_t _nParamlength,
    int32_t * _pTicket,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nCmdID | Command ID. |
| in | _bQuery | Whether command is query or not. |
| in | _pParam | The integer array parameter to the device command. |
| in | _nParamlength | The length of the integer array in elements. |
| out | _pTicket | Pointer to a variable receiving the internal ticket for the issued device command. |
| out | _phRsp | Pointer to a buffer receiving the response handle. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.12 Function ExecSingleIntParamCmd()

**Description** Execute a device command with one integer parameter.

**Prototype**

```
Res_t ExecSingleIntParamCmd(
    Conn_h _hConnection,
    uint32_t _nCmdID,
    int32_t _bQuery,
    int32_t _nParam,
    int32_t * _pTicket,
    Rsp_h * _phRsp
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nCmdID | Command ID. |
| in | _bQuery | Whether command is query or not. |
| in | _nParam | The integer parameter to the device command. |
| out | _pTicket | Pointer to a variable receiving the internal ticket for the issued device command. |
| out | _phRsp | Pointer to a buffer receiving the response handle. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.13 Function GetAbbeNumbers()

**Description** Query Abbe numbers from 0D7000

**Prototype**

```
Res_t GetAbbeNumbers(
    Conn_h _hConnection,
    const float ** _paAbbe,
    int32_t * _pnLayerCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _paAbbe | Array containing Abbe numbers returned by device, only used for synchronous connection. |
| out | _pnLayerCount | Layer count of being measured material (i.e. number of Abbe numbers), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.14 Function GetAnalogOutput()

**Description** Query analog output configuration from 0D7000

**Prototype**

```
Res_t GetAnalogOutput(
    Conn_h _hConnection,
    int32_t _nIndex,
    int32_t * _pnSignalID,
    float * _pnMin,
    float * _pnMax,
    float * _pnVolMin,
    float * _pnVolMax,
    float * _pnVolInvalid,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nIndex | Analog output index (0 for channel 1; 1 for channel 2). |
| out | _pnSignalID | ID of the signal sent through selected analog output, only used for synchronous connection. |
| out | _pnMin | Signal lower limit value, corresponds to minimum output voltage, only used for synchronous connection. |
| out | _pnMax | Signal upper limit value, corresponds to maximum output voltage, only used for synchronous connection. |
| out | _pnVolMin | Minimum output voltage, only used for synchronous connection. |
| out | _pnVolMax | Maximum output voltage, only used for synchronous connection. |
| out | _pnVolInvalid | Voltage corresponding to invalid measuring result, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.15 Function GetCCDRange()

**Description** Query CCD range from OD7000

**Prototype**

```
Res_t GetCCDRange(
    Conn_h _hConnection,
    int32_t * _pnStartPixel,
    int32_t * _pnStopPixel,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnStartPixel | Start pixel for the CCD range returned by device, only used for synchronous connection. |
| out | _pnStopPixel | Stop pixel for the CCD range returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.16 Function GetConfocalDetectionThreshold()

**Description** Query peak detection threshold for confocal mode from OD7000

**Prototype**

```
Res_t GetConfocalDetectionThreshold(
    Conn_h _hConnection,
    float * _pnThreshold,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnThreshold | Detection threshold used by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.17 Function GetConnectionOutputSignals()

**Description** Get IDs of output signal data for the current connection from OD7000

**Prototype**

```
Res_t GetConnectionOutputSignals(
    Conn_h _hConnection,
    const int32_t ** _pSignalIDbuffer,
    int32_t * _pSignalCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pSignalIDbuffer | A pointer to the array where the signal IDs are stored, only used for synchronous connection. |
| out | _pSignalCount | A pointer to an integer which stores the number of signals, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.18 Function GetDataAverage()

**Description** Query data average.

**Prototype**

```
Res_t GetDataAverage(
    Conn_h _hConnection,
    int32_t * _pnAVD,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnAVD | Data average returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.19 Function GetDetectionWindow()

**Description** Query windows where peaks can be detected from 0D7000

**Prototype**

```
Res_t GetDetectionWindow(
    Conn_h _hConnection,
    const float ** _panDWs,
    int32_t * _pnWindowCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _panDWs | A float array containing window definitions: window 1 left limit, window 1 right limit, window 2 left limit, window 2 right limit..., only used for synchronous connection. |
| out | _pnWindowCount | Number of windows (the size of detection window array should be twice the number of windows!), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.20 Function GetDetectionWindowActive()

**Description** Query detection window active state from 0D7000

**Prototype**

```
Res_t GetDetectionWindowActive(
    Conn_h _hConnection,
    int32_t * _pbActive,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pbActive | Active state returned by device: 1 - active; 0 - inactive, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.21 Function GetDeviceFirmwareVersion()

**Description** Query device firmware version from 0D7000

**Prototype**

```
Res_t GetDeviceFirmwareVersion(
    Conn_h _hConnection,
    const char ** _pstrVersion,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pstrVersion | Firmware version (a zero-terminated string) returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.22 Function GetDutyCycle()

**Description** Query duty cycle from 0D7000

**Prototype**

```
Res_t GetDutyCycle(
    Conn_h _hConnection,
    float * _pnDutyCycle,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnDutyCycle | Current duty cycle returned by the device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.23 Function GetEncoderCounter()

**Description** Query device current encoder position.

**Prototype**

```
Res_t GetEncoderCounter(
    Conn_h _hConnection,
    int32_t _nAxis,
    int32_t * _pnPosition,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAxis | The index of encoder axis to be queried |
| out | _pnPosition | Encoder position returned from the device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.24 Function GetEncoderCounterSource()

**Description** Query current encoder counter counting source from 0D7000

**Prototype**

```
Res_t GetEncoderCounterSource(
    Conn_h _hConnection,
    int32_t _nAxis,
    int32_t * _pnSource,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAxis | The index of encoder axis to be queried |
| out | _pnSource | Encoder counting source returned from device. (see A.15), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.25 Function GetEncoderTriggerEnabled()

**Description** Query current encoder trigger status from 0D7000

**Prototype**

```
Res_t GetEncoderTriggerEnabled(
    Conn_h _hConnection,
    int32_t * _pbEnabled,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pbEnabled | Encoder trigger enabled status returned from device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.26 Function GetEncoderTriggerProperty()

**Description** Query device current encoder trigger property.

**Prototype**

```
Res_t GetEncoderTriggerProperty(
    Conn_h _hConnection,
    int32_t * _pnEncoderAxis,
    int32_t * _pnStartPos,
    int32_t * _pnStopPos,
    float * _pnInterval,
    int32_t * _pbTriggerOnReturnMove,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection() |
| out | _pnEncoderAxis | Encoder axis index |
| out | _pnStartPos | Start position for encoder triggering, only used for synchronous connection. |
| out | _pnStopPos | Stop position for encoder triggering, only used for synchronous connection. |
| out | _pnInterval | Distance between adjacent triggering points in increments, only used for synchronous connection. |
| out | _pbTriggerOnReturnMove | Whether triggering is active during the return move from stop position to start position, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.27 Function GetFullScale()

**Description** Query device measuring scale.

**Prototype**

```
Res_t GetFullScale(
    Conn_h _hConnection,
    int32_t * _pnFullScale,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnFullScale | Measuring scale returned from device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.28 Function GetLampIntensity()

**Description** Query lamp intensity from OD7000

**Prototype**

```
Res_t GetLampIntensity(
    Conn_h _hConnection,
    float * _pnIntensity,
    Res_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection() |
| out | _pnIntensity | Lamp intensity in percent returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.29 Function GetLightSourceAutoAdapt()

**Description** Query device light source "Auto Adapt" mode settings from 0D7000

**Prototype**

```
Res_t GetLightSourceAutoAdapt(
    Conn_h _hConnection,
    int32_t * _pbAuto,
    float * _pnLevel,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pbAuto | "Auto Adapt" mode returned by device, only used for synchronous connection. |
| out | _pnLevel | CCD exposure level in percent returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.30 Function GetMeasuringMethod()

**Description** Query device measuring method from 0D7000.

**Prototype**

```
Res_t GetMeasuringMethod(
    Conn_h _hConnection,
    int32_t * _pnMethod,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnMethod | Measuring method returned from device: 0 - 0 - chromatic confocal 1 |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.31 Function GetMedian()

**Description** Query median width and percentile used for calculating sample median from 0D7000

**Prototype**

```
Res_t GetMedian(
    Conn_h _hConnection,
    int32_t * _pnMedianWidth,
    float * _pnPercentile,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection() |
| out | _pnMedianWidth | Median width returned by device, only used for synchronous connection. |
| out | _pnPercentile | Percentile returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.32 Function GetNumberOfPeaks()

**Description** Query number of the peaks to be detected from 0D7000

**Prototype**

```
Res_t GetNumberOfPeaks(
    Conn_h _hConnection,
    int32_t * _pnPeakNum,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnPeakNum | Number of peaks returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.33 Function GetOpticalProbe()

**Description** Query currently used optical probe in confocal measuring mode from OD7000

**Prototype**

```
Res_t GetOpticalProbe(
    Conn_h _hConnection,
    int32_t * _pnIndex,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnIndex | Optical probe index returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.34 Function GetPeakOrdering()

**Description** Query the sequence to order detected peaks from OD7000

**Prototype**

```
Res_t GetPeakOrdering(
    Conn_h _hConnection,
    int32_t * _pnPeakOrdering,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnPeakOrdering | Peak ordering sequence returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.35 Function GetRefractiveIndexTables()

**Description** Query currently used refractive index tables from 0D7000

**Prototype**

```
Res_t GetRefractiveIndexTables(
    Conn_h _hConnection,
    const int32_t ** _paTable,
    int32_t * _pnLayerCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _paTable | Array containing indices of currently used Refractive index table returned by device, only used for synchronous connection. |
| out | _pnLayerCount | Layer count of being measured materials (i.e. number of indices of refractive index table), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.36 Function GetRefractiveIndices()

**Description** Query refractive index from 0D7000

**Prototype**

```
Res_t GetRefractiveIndices(
    Conn_h _hConnection,
    const float ** _paRefIdx,
    int32_t * _pnLayerCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _paRefIdx | Array containing refractive indices returned by device, only used for synchronous connection. |
| out | _pnLayerCount | Layer count of being measured material (i.e. number of refractive indices), only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.37 Function GetScanRate()

**Description** Query scan rate from 0D7000

**Prototype**

```
Res_t GetScanRate(
    Conn_h _hConnection,
    float * _pnScanRate,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnScanRate | Scan rate in Hz returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.38 Function GetSpectrumAverage()

**Description** Query spectrum average from 0D7000

**Prototype**

```
Res_t GetSpectrumAverage(
    Conn_h _hConnection,
    int32_t * _pnAVS,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pnAVS | Spectrum average returned by device, only used for synchronous connection. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.39 Function ReadAbbeNumbersRsp()

**Description** Read in response of abbe number command.

**Prototype**

```
Res_t ReadAbbeNumbersRsp(
    Rsp_h _hRsp,
    const float ** _paAbbe,
    int32_t * _pnLayerCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _paAbbe | Array of Abbe numbers contained in the response. |
| out | _pnLayerCount | Layer count of being measured material (i.e. number of Abbe numbers) contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.40 Function ReadAnalogOutputRsp()

**Description** Read in response of analog output configuration command.

**Prototype**

```
Res_t ReadAnalogOutputRsp(
    Rsp_h _hRsp,
    int32_t * _pnIndex,
    int32_t * _pnSignalID,
    float * _pnMin,
    float * _pnMax,
    float * _pnVolMin,
    float * _pnVolMax,
    float * _pnVolInvalid
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnIndex | Analog output index (0 for channel 1; 1 for channel 2) contained in the response. |
| out | _pnSignalID | ID of the signal sent through selected analog output, contained in the response. |
| out | _pnMin | Signal lower limit value, corresponds to minimum output voltage, contained in the response. |
| out | _pnMax | Signal upper limit value, corresponds to maximum output voltage, contained in the response. |
| out | _pnVolMin | Minimum output voltage contained in the response. |
| out | _pnVolMax | Maximum output voltage contained in the response. |
| out | _pnVolInvalid | Voltage corresponding to invalid measuring result contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.41 Function ReadCCDRangeRsp()

**Description** Read in response of CCD range command.

**Prototype**

```
Res_t ReadCCDRangeRsp(
    Rsp_h _hRsp,
    int32_t * _pnStartPixel,
    int32_t * _pnStopPixel
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnStartPixel | Start pixel for the CCD range contained in the response. |
| out | _pnStopPixel | Start pixel for the CCD range in percent contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.42 Function ReadConfocalDetectionThresholdRsp()

**Description** Read in response of peak detection threshold command for confocal mode.

**Prototype**

```
Res_t ReadConfocalDetectionThresholdRsp(
    Rsp_h _hRsp,
    float * _pnThreshold
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnThreshold | Detection threshold contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.43 Function ReadConnectionOutputSignalRsp()

**Description** Read in response of connection output signal command.

**Prototype**

```
Res_t ReadConnectionOutputSignalRsp(
    Rsp_h _hRsp,
    const int32_t ** _pSignalIDbuffer,
    int32_t * _pSignalCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pSignalIDbuffer | A pointer to the array where the signal IDs are stored. |
| out | _pSignalCount | A pointer to an integer which stores the number of signals output for current connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.44 Function ReadDarkReferenceRsp()

**Description** Read in response of dark reference command.

**Prototype**

```
Res_t ReadDarkReferenceRsp(
    Rsp_h _hRsp,
    float * _pMinFrequency
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pMinFrequency | (Virtual) scan rate in Hertz at which the CCD would be saturated by the stray light (contained in the response). |
| return | | Zero on success. Negative: error or warning code. |

## C.45 Function ReadDataAverageRsp()

**Description** Read in response of data average command.

**Prototype**

```
Res_t ReadDataAverageRsp(
    Rsp_h _hRsp,
    int32_t * _pnAVD
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnAVD | Data average contained in response. |
| return | | Zero on success. Negative: error or warning code. |

## C.46 Function ReadDetectionWindowActiveRsp()

**Description** Read in response of detection window active command.

**Prototype**

```
Res_t ReadDetectionWindowActiveRsp(
    Rsp_h _hRsp,
    int32_t * _pbActive
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pbActive | Active state contained in the response: 1 - active; 0 - inactive. |
| return | | Zero on success. Negative: error or warning code. |

## C.47 Function ReadDetectionWindowRsp()

**Description** Read in response of detection window command.

**Prototype**

```
Res_t ReadDetectionWindowRsp(
    Rsp_h _hRsp,
    const float ** _panDWs,
    int32_t * _pnWindowCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| out | _panDWs | A float array containing window definitions: window 1 left limit, window 1 right limit, window 2 left limit, window 2 right limit..., contained in the response. |
| out | _pnWindowCount | Number of windows (the size of detection window array should be twice the number of windows), contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.48 Function ReadDeviceFirmwareVersionRsp()

**Description** Read in response of device firmware query command.

**Prototype**

```
Res_t ReadDeviceFirmwareVersionRsp(
    Rsp_h _hRsp,
    const char ** _pstrVersion
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pstrVersion | Firmware version (a zero-terminated string) contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.49 Function ReadDeviceTriggerModedRsp()

**Description** Read in response of device triggering mode setting command.

**Prototype**

```
Res_t ReadDeviceTriggerModedRsp(
    Rsp_h _hRsp,
    int32_t * _pnTriggerMode
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnTriggerMode | Device triggering mode contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.50 Function ReadDownloadDeviceSpectrumRsp()

**Description** Read in response of spectrum download command.

**Prototype**

```
Res_t ReadDownloadDeviceSpectrumRsp(
    Rsp_h _hRsp,
    int32_t * _pnSpecType,
    int32_t * _pnStartChannelIdx,
    int32_t * _pnChannelNumber,
    const int16_t ** _pSpectrumBuffer,
    int32_t * _pSpectrumLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnSpecType | Spectrum type contained in the response (definition see Appendix A.12). |
| out | _pnStartChannelIdx | Index of starting channel for which spectrum should be downloaded, contained in the response. |
| out | _pnChannelNumber | Number of channels to be downloaded, contained in the response. |
| out | _pSpectrumBuffer | A pointer to a short/word pointer where the spectra are stored, contained in the response. |
| out | _pSpectrumLength | Length of a spectrum for single channel, contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.51 Function ReadDownloadTableRsp()

**Description** Read in response of table download command.

**Prototype**

```
Res_t ReadDownloadTableRsp(
    Rsp_h _hRsp,
    int32_t * _pnTableType,
    int32_t * _pnTableIdx,
    const int8_t ** _pTable,
    int32_t * _nTableLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnTableType | Table type. |
| out | _pnTableIdx | Index of the downloaded table. |
| out | _pTable | A pointer to a byte pointer where the downloaded table is stored (managed by the library). |
| out | _nTableLength | Table length (in bytes). |
| return | | Zero on success. Negative: error or warning code. |

## C.52 Function ReadDutyCycleRsp()

**Description** Read in response of duty cycle command.

**Prototype**

```
Res_t ReadDutyCycleRsp(
    Rsp_h _hRsp,
    float * _pnDutyCycle
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnDutyCycle | Duty cycle contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.53 Function ReadEncoderCounterRsp()

**Description** Read in response of encoder counter command.

**Prototype**

```
Res_t ReadEncoderCounterRsp(
    Rsp_h _hRsp,
    int32_t * _pnAxis,
    int32_t * _pnPosition
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnAxis | The index of encoder axis contained in the response. |
| out | _pnPosition | Encoder position contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.54 Function ReadEncoderCounterSourceRsp()

**Description** Read in response of encoder counter source command.

**Prototype**

```
Res_t ReadEncoderCounterSourceRsp(
    Rsp_h _hRsp,
    int32_t * _pnAxis,
    int32_t * _pnSource
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnAxis | The index of encoder axis contained in the response. |
| out | _pnSource | Encoder counting source contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.55 Function ReadEncoderPreloadFunctionRsp()

**Description** Read in response of encoder preload function command.

**Prototype**

```
Res_t ReadEncoderPreloadFunctionRsp(
    Rsp_h _hRsp,
    int32_t * _pnAxis,
    int32_t * _pnPreloadValue,
    int32_t * _pnEncoderPreloadConfig,
    int32_t * _pnEncoderTriggerSource
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnAxis | The index of encoder axis contained in the response. |
| out | _pnPreloadValue | Preload encoder position contained in the response. |
| out | _pnEncoderPreloadConfig | Preload function settings contained in the response (see A.16). |
| out | _pnEncoderTriggerSource | Trigger source of the preload function contained in the response (see A.15). |
| return | | Zero on success. Negative: error or warning code. |

## C.56 Function ReadEncoderTriggerEnabledRsp()

**Description** Read in response of encoder trigger command.

**Prototype**

```
Res_t ReadEncoderTriggerEnabledRsp(
    Rsp_h _hRsp,
    int32_t * _pbEnabled
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pbEnabled | Encoder trigger enabled status contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.57 Function ReadEncoderTriggerPropertyRsp()

**Description** Read in response of encoder trigger property setting command.

**Prototype**

```
Res_t ReadEncoderTriggerPropertyRsp(
    Rsp_h _hRsp,
    int32_t * _pnEncoderAxis,
    int32_t * _pnStartPos,
    int32_t * _pnStopPos,
    float * _pnInterval,
    int32_t * _pbTriggerOnReturnMove
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnEncoderAxis | Encoder axis index contained in the response. |
| out | _pnStartPos | Start position for encoder triggering, contained in the response. |
| out | _pnStopPos | Stop position for encoder triggering, ocontained in the response. |
| out | _pnInterval | Distance between adjacent triggering points in increments, contained in the response. |
| out | _pbTriggerOnReturnMove | Whether triggering is active during the return move from stop position to start position, contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.58 Function ReadFullScaleRsp()

**Description** Read in response of measuring scale command.

**Prototype**

```
Res_t ReadFullScaleRsp(
    Rsp_h _hRsp,
    int32_t * _pnFullScale
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnFullScale | Measuring scale contained in response. |
| return | | Zero on success. Negative: error or warning code. |

## C.59 Function ReadLampIntensityRsp()

**Description** Read in response of lamp intensity command.

**Prototype**

```
Res_t ReadLampIntensityRsp(
    Rsp_h _hRsp,
    float * _pnIntensity
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnIntensity | Lamp intensity in percent contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.60 Function ReadLightSourceAutoAdaptRsp()

**Description** Read in response of light source "Auto Adapt" command.

**Prototype**

```
Res_t ReadLightSourceAutoAdaptRsp(
    Rsp_h _hRsp,
    int32_t * _pbAuto,
    float * _pnLevel
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pbAuto | "Auto Adapt" mode contained in the response. |
| out | _pnLevel | CCD exposure level in percent in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.61 Function ReadMeasuringMethodRsp()

**Description** Read in response of measuring method command.

**Prototype**

```
Res_t ReadMeasuringMethodRsp(
    Rsp_h _hRsp,
    int32_t * _pnMethod
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnMethod | Measuring method contained in the response: 0 - 0 - chromatic confocal 1 |
| return | | Zero on success. Negative: error or warning code. |

## C.62 Function ReadMedianRsp()

**Description** Read in response of median command.

**Prototype**

```
Res_t ReadMedianRsp(
    Rsp_h _hRsp,
    int32_t * _pnMedianWidth,
    float * _pnPercentile
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnMedianWidth | Median width contained in the response. |
| out | _pnPercentile | Percentile contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.63 Function ReadNumberOfPeaksRsp()

**Description** Read in response of peak number command.

**Prototype**

```
Res_t ReadNumberOfPeaksRsp(
    Rsp_h _hRsp,
    int32_t * _pnPeakNum
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnPeakNum | Number of peaks contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.64 Function ReadOpticalProbeRsp()

**Description** Read in response of optical probe selection command.

**Prototype**

```
Res_t ReadOpticalProbeRsp(
    Rsp_h _hRsp,
    int32_t * _pnIndex
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnIndex | Optical probe index contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.65 Function ReadPeakOrderingRsp()

**Description** Read in response of peak ordering sequence command.

**Prototype**

```
Res_t ReadPeakOrderingRsp(
    Rsp_h _hRsp,
    int32_t * _pnPeakOrdering
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnPeakOrdering | Peak ordering sequence contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.66 Function ReadRefractiveIndexTablesRsp()

**Description** Read in response of refractive index table command.

**Prototype**

```
Res_t ReadRefractiveIndexTablesRsp(
    Rsp_h _hRsp,
    const int32_t ** _paTable,
    int32_t * _pnLayerCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _paTable | Array of indices of currently used Refractive index table contained in the response. |
| out | _pnLayerCount | Layer count of being measured material (i.e. number of Abbe numbers) contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.67 Function ReadRefractiveIndicesRsp()

**Description** Read in response of refractive index command.

**Prototype**

```
Res_t ReadRefractiveIndicesRsp(
    Rsp_h _hRsp,
    const float ** _paRefIdx,
    int32_t * _pnLayerCount
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _paRefIdx | Array of refractive indices contained in the response. |
| out | _pnLayerCount | Layer count of being measured material (i.e. number of refractive indices) contained in the response. |
| return | | Zero on success. Negative: error or warning code. |

## C.68 Function ReadScanRateRsp()

**Description** Read in response of scan rate command.

**Prototype**

```
Res_t ReadScanRateRsp(
    Rsp_h _hRsp,
    float * _pnScanRate
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnScanRate | Scan rate in Hz contained in response. |
| return | | Zero on success. Negative: error or warning code. |

## C.69 Function ReadSingleFloatArrayParamRsp()

**Description** Extract a single float array parameter from the response to a device command.

**Prototype**

```
Res_t ReadSingleFloatArrayParamRsp(
    Rsp_h _hRsp,
    uint32_t _nCmdID,
    uint32_t _nIdx,
    const float ** _pParam,
    int32_t * _pnLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nCmdID | Command ID. |
| in | _nIdx | Parameter index. |
| out | _pParam | Pointer to a variable receiving the pointer to the float array data. |
| out | _pnLength | Pointer to an integer variable receiving the array length in number of elements. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.70 Function ReadSingleFloatParamRsp()

**Description** Extract a single float parameter from the response to a device command.

**Prototype**

```
Res_t ReadSingleFloatParamRsp(
    Rsp_h _hRsp,
    uint32_t _nCmdID,
    uint32_t _nIdx,
    float * _pParam
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nCmdID | Command ID. |
| in | _nIdx | Parameter index. |
| out | _pParam | Pointer to a float variable receiving the parameter value. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.71 Function ReadSingleIntArrayParamRsp()

**Description** Extract a single integer array parameter from the response to a device command.

**Prototype**

```
Res_t ReadSingleIntArrayParamRsp(
    Rsp_h _hRsp,
    uint32_t _nCmdID,
    uint32_t _nIdx,
    const int32_t ** _pParam,
    int32_t * _pnLength
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nCmdID | Command ID. |
| in | _nIdx | Parameter index. |
| out | _pParam | Pointer to a variable receiving the pointer to the integer array data. |
| out | _pnLength | Pointer to an integer variable receiving the array length in number of elements. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.72 Function ReadSingleIntParamRsp()

**Description** Extract a single integer parameter from the response to a device command.

**Prototype**

```
Res_t ReadSingleIntParamRsp(
    Rsp_h _hRsp,
    uint32_t _nCmdID,
    uint32_t _nIdx,
    int32_t * _pParam
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by ExecCommand() or passed through response and update callback function. |
| in | _nCmdID | Command ID. |
| in | _nIdx | Parameter index. |
| out | _pParam | Pointer to an integer variable receiving the parameter value. |
| return | | Standard API result code. Use the CHR_SUCCESS macro to check for success. |

## C.73 Function ReadSpectrumAverageRsp()

**Description** Read in response of spectrum average command.

**Prototype**

```
Res_t ReadSpectrumAverageRsp(
    Rsp_h _hRsp,
    int32_t * _pnAVS
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hRsp | Response handle, either returned by "ExecCommand" or passed through response and update callback function. |
| out | _pnAVS | Spectrum average contained in response. |
| return | | Zero on success. Negative: error or warning code. |

## C.74 Function SaveCurrentSettingInDevice()

**Description** Send command to save current setting in device.

**Prototype**

```
Res_t SaveCurrentSettingInDevice(
    Conn_h _hConnection,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.75 Function SetAbbeNumbers()

**Description** Set Abbe numbers.

**Prototype**

```
Res_t SetAbbeNumbers(
    Conn_h _hConnection,
    const float * _paAbbe,
    int32_t _nLayerCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _paAbbe | Array containing Abbe numbers (from 0 to 500). |
| in | _nLayerCount | Layer count of being measured material (i.e. number of Abbe numbers to be set). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.76 Function SetAnalogOutput()

**Description** Set analog output configuration.

**Prototype**

```
Res_t SetAnalogOutput(
    Conn_h _hConnection,
    int32_t _nIndex,
    int32_t _nSignalID,
    float _nMin,
    float _nMax,
    float _nVolMin,
    float _nVolMax,
    float _nVolInvalid,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nIndex | Analog output index (0 for channel 1; 1 for channel 2). |
| in | _nSignalID | ID of the signal to be sent through selected analog output. |
| in | _nMin | Signal lower limit value, corresponds to minimum output voltage. |
| in | _nMax | Signal upper limit value, corresponds to maximum output voltage. |
| in | _nVolMin | Minimum output voltage. |
| in | _nVolMax | Maximum output voltage. |
| in | _nVolInvalid | Voltage corresponding to invalid measuring result. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.77 Function SetCCDRange()

**Description** Set CCD range.

**Prototype**

```
Res_t SetCCDRange(
    Conn_h _hConnection,
    int32_t _nStartPixel,
    int32_t _nStopPixel,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nStartPixel | Start pixel for the CCD range. |
| in | _nStopPixel | Stop pixel for the CCD range. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.78 Function SetConfocalDetectionThreshold()

**Description** Set peak detection threshold for confocal mode.

**Prototype**

```
Res_t SetConfocalDetectionThreshold(
    Conn_h _hConnection,
    float _nThreshold,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nThreshold | Detection threshold (depends on the sensor model). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.79 Function SetConnectionOutputSignals()

**Description** Order signal data from 0D7000

**Prototype**

```
Res_t SetConnectionOutputSignals(
    Conn_h _hConnection,
    const int32_t * _anSignals,
    int32_t _nSignalsCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _anSignals | An array of integer numbers indicating the signals to be included in the data stream. |
| in | _nSignalsCount | Number of signals (array length). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.80 Function SetDataAverage()

**Description** Set data average.

**Prototype**

```
Res_t SetDataAverage(
    Conn_h _hConnection,
    int32_t _nAVD,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAVD | Data average (from 1 to 999). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.81 Function SetDetectionWindow()

**Description** Set windows where peaks can be detected.

**Prototype**

```
Res_t SetDetectionWindow(
    Conn_h _hConnection,
    const float * _anDWs,
    int32_t _nWindowCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _anDWs | A float array containing window definitions: window 1 left limit, window 1 right limit, window 2 left limit, window 2 right limit... |
| in | _nWindowCount | Number of windows (the size of detection window array should be twice the number of windows!). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.82 Function SetDetectionWindowActive()

**Description** Enable or disable limits/detection window for peak detection under confocal

**Prototype**

```
Res_t SetDetectionWindowActive(
    Conn_h _hConnection,
    int32_t _bActive,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _bActive | 1: limit active; 0: limit inactive. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.83 Function SetDeviceTriggerMode()

**Description** Set device triggering mode.

**Prototype**

```
Res_t SetDeviceTriggerMode(
    Conn_h _hConnection,
    int32_t _nTriggerMode,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nTriggerMode | Device triggering mode. For details see Appendix A.14. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.84 Function SetDutyCycle()

**Description** Set duty cycle.

**Prototype**

```
Res_t SetDutyCycle(
    Conn_h _hConnection,
    float _nDutyCycle,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nDutyCycle | Duty cyle (0..49 100). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.85 Function SetEncoderCounter()

**Description** Set the encoder position.

**Prototype**

```
Res_t SetEncoderCounter(
    Conn_h _hConnection,
    int32_t _nAxis,
    int32_t _nPosition,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAxis | The index of encoder axis to be set |
| in | _nPosition | Encoder position to be set. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.86 Function SetEncoderCounterSource()

**Description** Set the encoder counter counting source.

**Prototype**

```
Res_t SetEncoderCounterSource(
    Conn_h _hConnection,
    int32_t _nAxis,
    int32_t _nSource,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAxis | The index of encoder axis to be set |
| in | _nSource | Encoder counting source to be set (see A.15). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.87 Function SetEncoderPreloadFunction()

**Description** Set the encoder preload function.

**Prototype**

```
Res_t SetEncoderPreloadFunction(
    Conn_h _hConnection,
    int32_t _nAxis,
    int32_t _nPreloadValue,
    int32_t _nEncoderPreloadConfig,
    int32_t _nEncoderTriggerSource,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAxis | Encoder axis index to be set |
| in | _nPreloadValue | Encoder position to be preloaded. |
| in | _nEncoderPreloadConfig | Preload function settings (see A.16). |
| in | _nEncoderTriggerSource | Trigger source of the preload function (see A.15). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.88 Function SetEncoderTriggerEnabled()

**Description** Enable encoder trigger.

**Prototype**

```
Res_t SetEncoderTriggerEnabled(
    Conn_h _hConnection,
    int32_t _bEnabled,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _bEnabled | Enable encoder trigger, otherwise device is triggered by synchronization signal. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.89 Function SetEncoderTriggerProperty()

**Description** Set the property for encoder triggering.

**Prototype**

```
Res_t SetEncoderTriggerProperty(
    Conn_h _hConnection,
    int32_t _nEncoderAxis,
    int32_t _nStartPos,
    int32_t _nStopPos,
    float _nInterval,
    int32_t _bTriggerOnReturnMove,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nEncoderAxis | Encoder axis index |
| in | _nStartPos | Start position for encoder triggering. |
| in | _nStopPos | Stop position for encoder triggering. |
| in | _nInterval | Distance between adjacent triggering points in increments. |
| in | _bTriggerOnReturnMove | Whether triggering is active during the return move from stop position to start position. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.90 Function SetLampIntensity()

**Description** Set lamp intensity.

**Prototype**

```
Res_t SetLampIntensity(
    Conn_h _hConnection,
    float _nIntensity,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nIntensity | Lamp intensity in percent. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.91 Function SetLightSourceAutoAdapt()

**Description** Set device light source "Auto Adapt" mode related settings.

**Prototype**

```
Res_t SetLightSourceAutoAdapt(
    Conn_h _hConnection,
    int32_t _bAuto,
    float _nLevel,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _bAuto | "Auto Adapt" mode on/off: 1 - on; 0 - off. |
| in | _nLevel | CCD exposure level in percent (from 0 to 100), only if "Auto Adapt" mode is on. |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.92 Function SetMeasuringMethod()

**Description** Set device measuring method

**Prototype**

```
Res_t SetMeasuringMethod(
    Conn_h _hConnection,
    int32_t _nMethod,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nMethod | Measuring method: 0 - chromatic confocal 1 . |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.93 Function SetMedian()

**Description** Set median width and percentile used for calculating sample median.

**Prototype**

```
Res_t SetMedian(
    Conn_h _hConnection,
    int32_t _nMedianWidth,
    float _nPercentile,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection() |
| in | _nMedianWidth | Median width (from 0 to 255) |
| in | _nPercentile | Percentile |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.94 Function SetNumberOfPeaks()

**Description** Set number of the peaks to be detected.

**Prototype**

```
Res_t SetNumberOfPeaks(
    Conn_h _hConnection,
    int32_t _nPeakNum,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nPeakNum | Number of peaks (from 1 to 16). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.95 Function SetOpticalProbe()

**Description** Select optical probe in confocal measuring mode.

**Prototype**

```
Res_t SetOpticalProbe(
    Conn_h _hConnection,
    int32_t _nIndex,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nIndex | Optical probe index (from 0 to 15). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.96 Function SetRefractiveIndexTables()

**Description** Select refractive index tables to use.

**Prototype**

```
Res_t SetRefractiveIndexTables(
    Conn_h _hConnection,
    const int32_t * _paTable,
    int32_t _nLayerCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _paTable | Array containing indices of refractive index table (from 0 to 16). 0: no special table used |
| in | _nLayerCount | Layer count of being measured materials (i.e. number of indices of refractive index table). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.97 Function SetRefractiveIndices()

**Description** Set refractive index.

**Prototype**

```
Res_t SetRefractiveIndices(
    Conn_h _hConnection,
    const float * _paRefIdx,
    int32_t _nLayerCount,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _paRefIdx | Array containing refractive indices. |
| in | _nLayerCount | Layer count of being measured material (i.e. number of refractive indices to be set). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.98 Function SetScanRate()

**Description** Set device scan rate.

**Prototype**

```
Res_t SetScanRate(
    Conn_h _hConnection,
    float _nScanRate,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection() |
| in | _nScanRate | Scan rate in Hz |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.99 Function SetSpectrumAverage()

**Description** Set spectrum average.

**Prototype**

```
Res_t SetSpectrumAverage(
    Conn_h _hConnection,
    int32_t _nAVS,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| in | _nAVS | Spectrum average (from 1 to 999). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.100 Function StartDataStream()

**Description** Start data stream from device.

**Prototype**

```
Res_t StartDataStream(
    Conn_h _hConnection,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection(). |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

## C.101 Function StopDataStream()

**Description** Stop data stream from device.

**Prototype**

```
Res_t StopDataStream(
    Conn_h _hConnection,
    int32_t * _pTicket
);
```

**Arguments**

| | | |
|---|---|---|
| in | _hConnection | Connection handle returned by OpenConnection() or OpenSharedConnection() |
| out | _pTicket | A pointer to an integer variable where the internal ticket for command is stored, only used for asynchronous connection. |
| return | | Zero on success. Negative: error or warning code. |

# Appendix D

# OD7000Lib Internal Commands

Following commands are the internal commands of OD7000Lib, user can pass these commands to the library through ExecCommand() or ExecCommandAsync(). The library will translate these commands to the corresponding device commands and forward to OD7000 device. The response of the commands, which are delivered to the user, also have the form of the internal commands. The purpose of these internal commands is to simplify or unify the usage of some device commands. The lowest byte of the command ID of these interal commands is always "0x25", the corresponding char is "%". If the command string is passed over to NewCommandFromString(), please do not add "$" in front of the command name as for the device command.

## D.1 Encoder Counter Value Command

**Description** Set/get the encoder counter value

**Command ID** CmdID_Encoder_Counter: 0x53504525

**Command Name** "%EPS".

**Command Format**

Set command: %EPS <arg0> <arg1>

Query command: %EPS? <arg0>

Response: %EPS(?) <arg0> <arg1>

**Parameter**

int32_t arg0: Axis index;

int32_t arg1: Encoder counter position;

**More Details** This command is to set the encoder count of the OD7000 device immediately.

## D.2 Encoder Counter Source Command

**Description** Set/get the input source of the encoder counter

**Command ID** CmdID_Encoder_Counter_Source: 0x53434525

**Command Name** "%ECS".

**Command Format**

Set command: %ECS <arg0> <arg1>

Query command: %ECS? <arg0>

Response: %ECS(?) <arg0> <arg1>

**Parameter**

int32_t arg0: Axis index;

int32_t arg1: Encoder counter source;

**More Details** This command sets the input source for the encoder counter of the given axis. The possible value of encoder counter source please check A.15.

## D.3 Encoder Preload function Command

**Description** Set the property related to encoder counter preload function

**Command ID** CmdID_Encoder_Preload_Function: 0x46504525

**Command Name** “%EPF”.

**Command Format**

Command: %EPF <arg0> <arg1> <arg2> <arg3>

Response: %EPF <arg0> <arg1> <arg2> <arg3>

**Parameter**

int32_t arg0: Axis index;

int32_t arg1: Encoder counter position to be preloaded;

int32_t arg2: Property of preload function;

int32_t arg3: Trigger source of the preload function;

**More Details** Apart from immediately setting the value of encoder counter, user can also set the encoder counter to be changed/loaded upon a certain "preload" event. The property of the preload event can be combined with the value defined in A.16. The possible trigger source of the event please refer to A.15.

## D.4 Encoder Trigger Enable Command

**Description** Set/get encoder triggle enable status

**Command ID** CmdID_Encoder_Trigger_Enabled = 0x45544525;

**Command Name** “%ETE”.

**Command Format**

Set command: %ETE <arg0>

Query command: %ETE?

Response: %ETE(?) <arg0>

**Parameter**

int32_t arg0: 1: Enable encoder trigger, 0: device is triggered by the synchronization signal;

**More Details** When OD7000 device is in trigger mode (either “trigger each” or “wait for trigger” mode), it can be triggered either by the synchronization signal or by the encoder. This command is to (de)select encoder as the trigger source.

## D.5 Encoder Trigger Property Command

**Description** Set/get the property related to encoder trigger

**Command ID** CmdID_Encoder_Trigger_Property: 0x50544525;

**Command Name** “%ETP”.

**Command Format**

Set command: %ETP <arg0> <arg1> <arg2> <arg3> <arg4>

Query command: %ETP?

Response: %ETP(?) <arg0> <arg1> <arg2> <arg3> <arg4>

**Parameter**

int32_t arg0: Encoder axis index which is used for triggering;

int32_t arg1: Start position for encoder triggering;

int32_t arg2: Stop position for encoder triggering;

float arg3: Distance between adjacent triggering points in increments;

int32_t arg4: 1: triggering is active; 0: trigger is inactive during the return move from stop position to start position;

**More Details** If the encoder is selected to be the trigger source of the device, user can use this command to set the property of the encoder trigger. Note, the query command can only be executed for synchronous connection.

## D.6 Device Trigger Mode Command

**Description** Set 0D7000 device trigger mode **Command**

**ID** CmdID_Device_Trigger_Mode: 0x4d525425; **Command**

**Name** "%TRM".

**Command Format**

Set command: %TRM <arg0>

Response: %TRM <arg0>

**Parameter**

int32_t arg0: triggering mode;

**More Details** 0D7000 device can be in four different trigger modes: "free-run", "trigger each", "wait for trigger" and "trigger window" mode. User can use this command to select the mode he needs. The exact meaning of each mode, please check the documentation of the device. The corresponding value of the modes is defined in A.14.

# Appendix E

# C# Wrapper

Inside the folder “cs”, there are API Wrapper files for c#. File “OD7000LibFunctionWrapper.cs” simply translates above API functions into c# style. “OD7000LibErrorDefWrapper.cs” contains all the defined error code, user can compare them to the return-values of API functions. “OD7000LibCmdWrapper.cs” and “OD7000LibConnectionWrapper.cs” provide convenient wrapper classes.

## E.1 TCHRLibCmdWrapper

Classes and functions defined in the class “TCHRLibCmdWrapper” in file “OD7000LibCmdWrapper.cs” are used for facilitating command sending and response reading.

**TCommand** General command class. User can create any “TCommand” object with either command ID (defined in “OD7000LibFunctionWrapper.cs”) or three/four letter command name (defined in the device manual). After creation, parameters of different types can be added into the command object with the member function “AddParameter”.

**TResponse** General response class, helps to read command response. It is created with the response handle and related informations, either directly given by funciton “ExecCommand” inside “OD7000LibFunc-tionWrapper.cs”, or the callback function ResponseAndUpdateCallback(). Property “Param-Count”, functions “GetParameter”, “GetIntParameter”... can be used to read response parame-ters.

**Specific Command Class** Predefined command classes for frequent used commands, for example “TMeasuringMethodCmd” class for setting/getting measuring method, “TScanRateCmd” class for setting/getting scan rate... The constructor of the classes already contain the related parameters

**Specific Response Class** Predefined response classes for frequent used commands, for example “TMeasuringMethodRsp”, “TScanRateRsp”... Every specific response class contains related parameters as properties.

**ExecCommand(Async/String)** Command execution functions which take the object of “TCommand” or any specific command class as input command. “_oLibRsp” returned by the synchronous command function “ExecCommand” or “ExecCommandString” can be directly casted into corresponding response class, like “TScanRateRsp”.

## E.2 TCHRLibConnectionWrapper

Class “TCHRLibConnectionWrapper” defined in file “OD7000LibConnectionWrapper.cs” wrapps the single connection, simplifies data reading and the usage of callback functions.

**TCHRLibConnData** Read in data, which is stored in the buffer returned by API function GetNextSamples(), GetLastSample() or “SampleDataCallback”. This class already does the job of data marshaling von unmanaged memory. It returns single data in double format.

**TCHRLibConnectionWrapper** Wrapps single connection. Contained member function “ExecCommandAsync” takes directly the object of “TCommand” (or specific command class) as input command and returns the object of response classes defined in “TCHRLibCmdWrapper”. Its data acquisition functions “GetNextSamples” and “GetLastSample” gives back “TCHRLibConnData”. The defined callback function “ConnResponseAndUpdateCallback” and “ConnSampleDataCallback” also use c# class “TResponse” and “TCHRLibConnData”. The related processing of API C interface is already done inside of the class, which largely simplify the usage of API functions in managed environment.