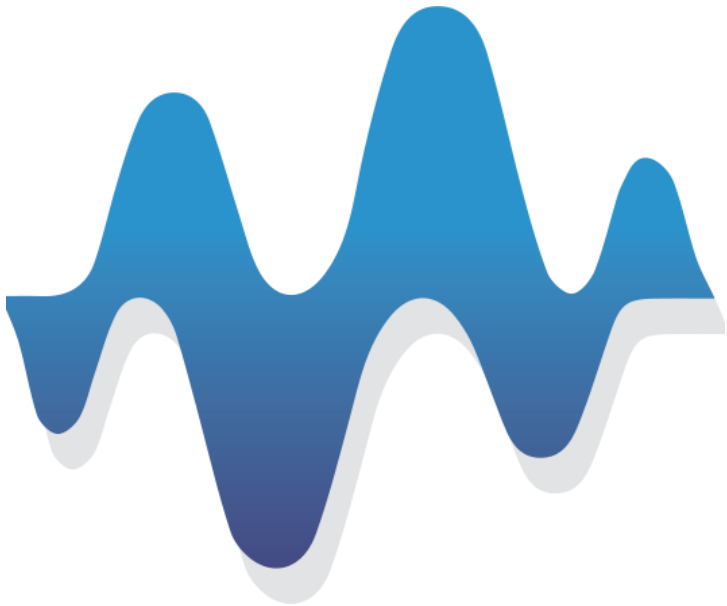
Ocean Insight

OceanDirect

Programming Manual

For product: OceanDirect



Locations

Americas

Manufacturing & Logistics
3500 Quadrangle Blvd., Orlando, FL 32817, USA

Sales: info@oceaninsight.com
Orders: orders@oceaninsight.com
Support: techsupport@oceaninsight.com

Phone: +1 727.733.2447
Fax: +1 727.733.3962

Europe, Middle East & Africa

Sales & Support
Geograaf 24, 6921 EW Duiven, The Netherlands

Manufacturing & Logistics
Maybaachstrasse 11, 73760 Ostfildern, Germany

Email: info@oceaninsight.eu

Netherlands: +31 26-319-0500
Netherlands Fax: +31 26-319-0505
Germany: +49 711-341696-0
UK: +44 1865-819922
France: +33 442-386-588

Asia

Ocean Insight China
666 Gubei Rd., Kirin Tower Suite 601B
Changning District, Shanghai, PRC, 200336

Email: asiasales@oceaninsight.com

China: +86 21-6295-6600
China Fax: +86 21-6295-6708
Japan & Korea: +82 10-8514-3797

Ocean Insight India
Prestige Shantiniketan, Gate no.2
Tower C, 7th Floor
Whitefield main road, Mahadevpura
Bengaluru-560048 Karnataka, India

Phone: +91 80-67475336

www.oceaninsight.com

Table of Contents

Introduction

OceanDirect™ is a powerful Software Developer's Kit (SDK) that allows you to easily write custom software solutions for your Ocean Insight spectrometers. An Application Programming Interface (API) provides functions to communicate with and control Ocean Insight spectrometers. With this product you can connect to spectrometers, set acquisition parameters such as integration time, and acquire spectra. By integrating OceanDirect into your own software application, you have complete control over spectrometers and devices.

This document discusses the capabilities of OceanDirect and provides high level information on the data structures. Detailed information on the data structures is documented within the SDK. In addition, sample code using the SDK may be found on the Ocean Insight website.

OceanDirect was developed in the C++ language and includes native libraries for Windows operating systems.

Operating System Support

- Windows: Windows 10 or higher

Language Support

You can develop OceanDirect-based applications in the following languages:

- C/C++/C#/Visual Basic (Microsoft Visual Studio environment)
- C (standard interface environment)
- LabVIEW (Windows only, Version 8 or greater)
- MATLAB
- Python (version 3 or later)

OceanDirect Spectrometers

OceanDirect is spectrometer-independent; the same function calls work for all spectrometers.

The following Ocean Insight spectrometers are supported by OceanDirect:

- Flame
- HR4000
- HR4Pro
- NIRQuest
- NIRQuest+
- QE Pro
- STS
- Ocean HDX
- Ocean FX

Development Environments

Windows Development

For purposes of programming in Windows, you can access OceanDirect functionality via two DLL files:

- OceanDirect.dll contains the functions that allow you to control all spectrometer settings and acquire spectra. For example, you can set the integration time, scans to average, enable electric dark correction, etc.
- NetOceanDirect.dll is a similar DLL but specifically for development using the Microsoft .NET Framework.

LabVIEW Development

For LabVIEW developers, OceanDirect provides a set of VI files that expose its functionality in a fashion that is natural to the LabVIEW development environment. Behind the scenes, these VIs invoke the .NET methods contained in the DLL files that comprise OceanDirect.

MATLAB Development

For MATLAB developers, OceanDirect provides an 'm' file script that exposes its functionality in a fashion that is natural to the MATLAB development environment. Behind the scenes, these scripts invoke the .NET methods contained in the DLL files that comprise OceanDirect.

Installation

Upon purchasing the software, you will be provided a link for downloading the software and an associated license key. Click on the link to access the installation file. If you did not receive the link, or have misplaced it, request a replacement email via the technical support request form.

When the installation process is finished, the following subdirectories will be created beneath the OceanDirect "home" directory:

Subdirectory	Contents
doc	Documentation relating to OceanDirect and its API
include	Header files for use with C/C++ application development
lib	Libraries for client applications

Once you have installed the software, you'll want to verify your installation, look at the samples located at the at the OceanDirect product page at OceanInsight.com to get an idea of how the objects and methods for OceanDirect are organized and then run a sample program.

Installing OceanDirect Software

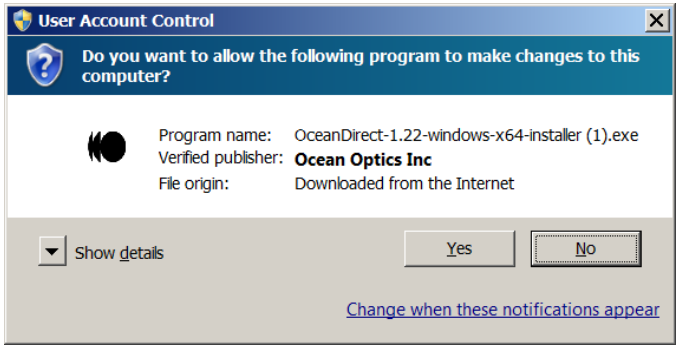
Simply download the file and double-click it in Windows Explorer to begin the installation procedure. The installer will guide you through the install process.

NOTE

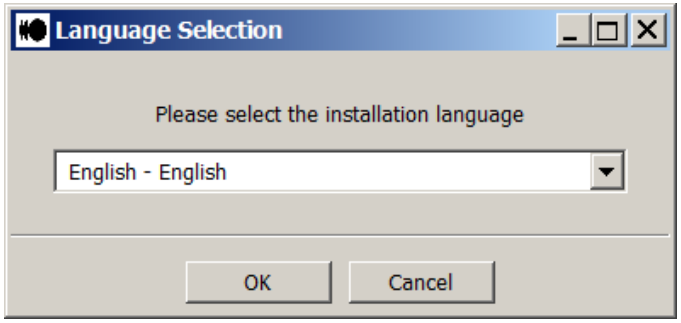
The computer on which you are installing the software must be connected to the Internet to validate the license key.

1. Start your Internet browser.
2. Navigate to the link provided to you and select the OceanDirect software (e.g., OceanDirect-x.xx-windows-64-installer.exe).
3. Save the software installer to the desired location.
4. Double-click on the file. The installer wizard guides you through the installation process. The default installation directory is c:\Program Files\Ocean Insight\OceanDirect SDK.

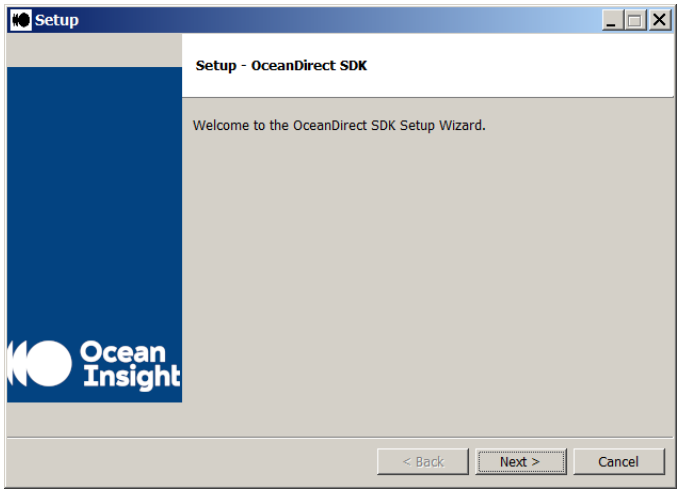
a. Allow the installation to begin by clicking “Yes.”



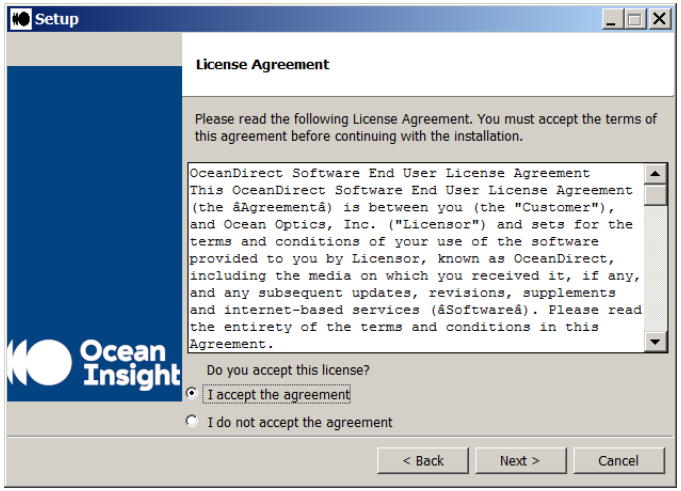
b. Choose your preferred language and click “OK.”



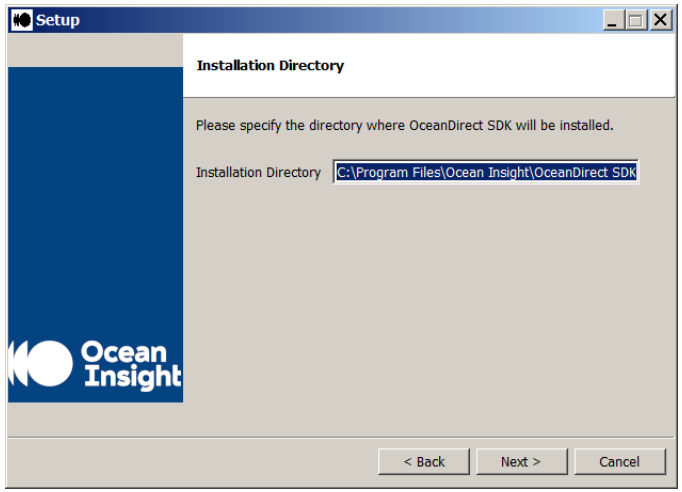
c. The Setup window is displayed. Click “Next” to continue.



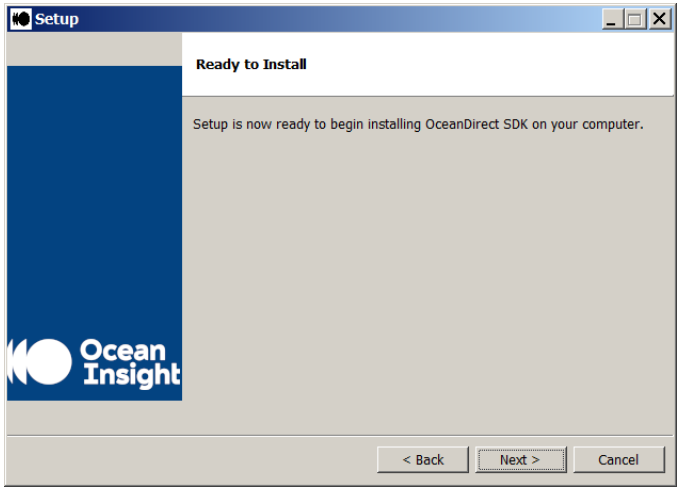
d. Review the License Agreement and select the “I accept the agreement” button, then click “Next.”



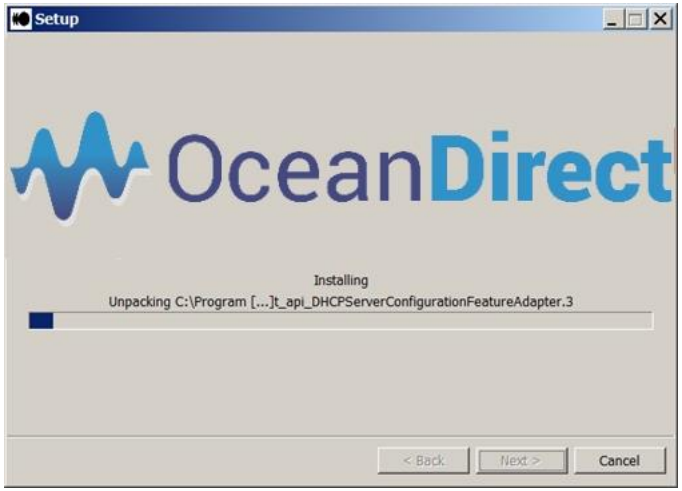
e. Choose the directory in which the software will be installed.



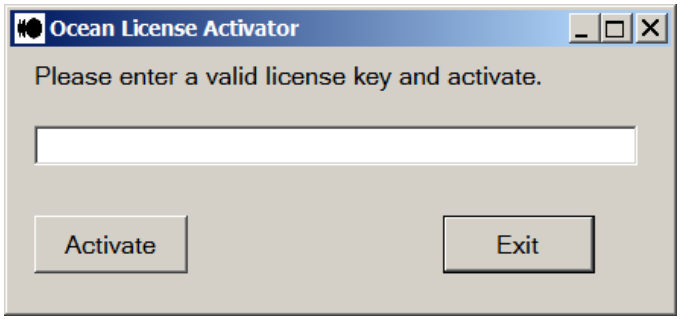
f. You are now ready to being the installation. Click “Next” to continue.



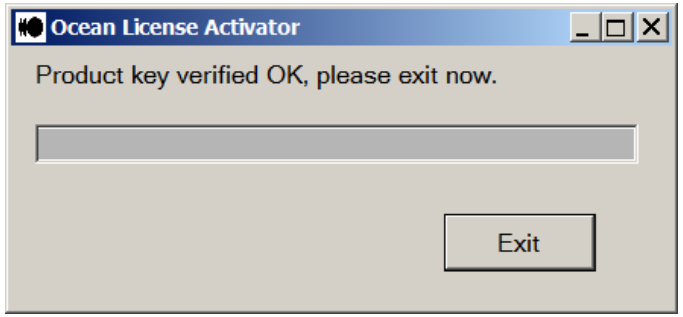
g. A progress bar is displayed showing the status of the installation.



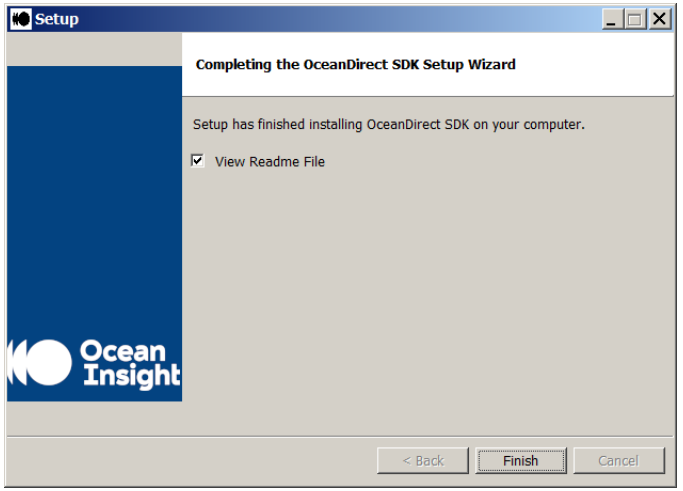
h. You will be prompted to enter your license key. Type the license number in and click on “Activate”.



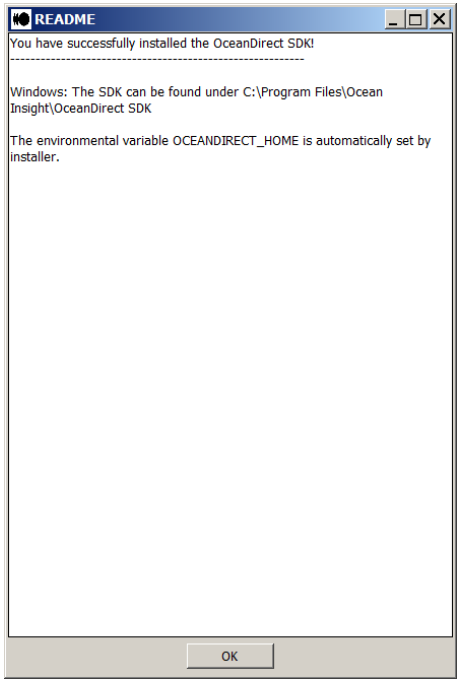
i. Once the license key has been verified, click on the “Exit” button.



j. Upon completion of license verification, you have the option of reviewing the Readme file. Click “Finish” to conclude.



k. If chosen, the Readme file is displayed. Click “OK” to close the window.



NOTE

If you do not enter the license key during installation, you will not be able to successfully use the program. An error code will be returned for all function calls indicating that the license is not active.

To activate the license at a later time, use the license activator tool “OceanLicenseActivator.exe” located in the directory in which you installed the software. Run this tool and it will prompt you for the license key that will activate the software.

Installing the Spectrometer Driver Software

NOTE

Do not plug your spectrometer in until after you have finished installing the OceanDirect software on your system.

Each spectrometer has unique steps for installing its drivers. Refer to your spectrometer’s installation guide for details on how to install the driver software.

Installation Troubleshooting

Problem	Possible Cause(s)	Suggested Solution(s)
You have installed the latest version of OceanDirect while your computer is plugged in to your spectrometer, but your application does not see it.	The old driver (ezusb.sys) for your spectrometer must be uninstalled.	Uninstall the old spectrometer driver.
You have installed the latest version of OceanDirect and plugged into your spectrometer, but your application does not see it.	The USB device needs to be enumerated.	Unplug the USB cable. Wait 5 seconds. Plug the USB cable back in.

Basic Sequence of Operations

This chapter describes the typical sequence of operations that the application must perform to control a spectrometer and acquire spectra. The functions described here are common to all Ocean Insight spectrometers.

This chapter provides generic syntax for C# method calls, created in the .net development environment. Refer to the sample code at the OceanDirect product page at OceanInsight.com for specific examples for additional languages.

There are also a number of optional features offered by some, but not all, spectrometers. Refer to Advanced Features for more information on these.

Further in the document, the Developing Your Application section discusses how to set up your development environment prior to creating your application.

Create an Instance of the Object

Before you can control your spectrometer, you must create an instance of the object. This is your gateway into all of the capabilities of the spectrometer.

NOTE

Your application must create only ONE instance of the object. This object is then shared by all spectrometers under the control of your application. All threads created by your application must then share a reference to the same instance of the object. Furthermore, there may only be ONE application (EXE) running on your computer that creates an object and controls spectrometers. All interaction with all Ocean Insight spectrometers attached to your computer must be performed within this single executable/application.

After importing the library, simply declare an instance for different languages. In the example below, we use “ocean” as the name of the instantiated object. The example in the function descriptions in this section are shown are in C#. Refer to the sample code on the product page for additional languages.

```
var ocean = OceanDirect.getInstance();
```

Find All Spectrometers

Next, you want to discover any spectrometers that are connected, either by USB or Ethernet, by using the findDevices method. The *findDevices()* method returns a tracked handle to a list of Devices objects containing metadata for all detected/known devices on USB and/or Ethernet.

Device Information returned includes: ID, name, if the device is in use, bus type, port number, IP address, and spectrum length.

```
Devices[] devs = ocean.findDevices();
if (0 == devs.Length)
{
Console.WriteLine("Nothing attached - press any key to exit!");
return;
}
```

Get Device ID

Each device found with *findDevices()* has a device ID, a number which is used in the function calls to address a given spectrometer, particularly if multiple spectrometers are attached. The default device ID returned if one spectrometer is attached is 2. Note that the device array is 0 based, so the first value in the array is at position 0.

```
int deviceID = 0;
int firstDevice = 0;
deviceID = devs[firstDevice].Id; // Get device ID of the device
```

Open Spectrometer

A spectrometer may be opened with the *ocean.openDevice(int deviceID, int% errorCode)*, which then allows operations to be performed on it. After being opened, the spectrometer should not be opened again until it is first closed using *ocean.closeDevice* (*int deviceID, int% errorCode*). The errorCode can be examined after the function is called. It will remain 0 if the function succeeds as the comment below indicates. In general, the errorCode can be tested after every function call.

```
int errorCode = 0;  // This variable can be tested after the function
```

Acquire a Spectrum

Now you are ready to acquire a spectrum. A spectrum is simply a one-dimensional array of pixel values, stored as “doubles”. The number of elements in this array varies for each spectrometer.

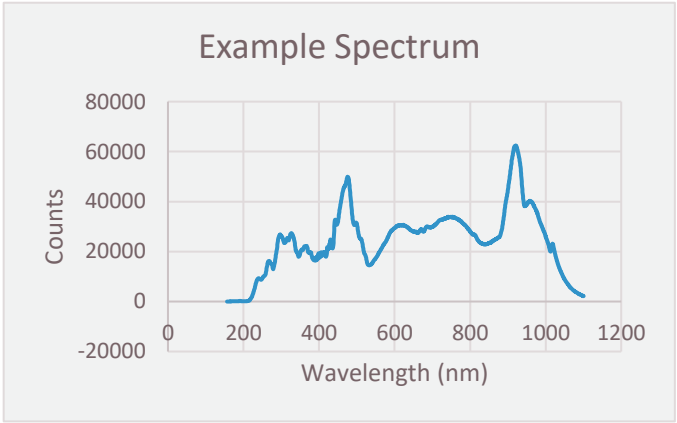
The function call is *getSpectrum(int deviceID, int% errorCode).*

```
double[] spectrum = ocean.getSpectrum(deviceId, ref errorCode);
```

When you call the *getSpectrum*()method, the spectrometer will return the next available spectrum to your application, assuming the spectrometer is in its default normal mode. The amount of time for the function to return may vary based on when the spectrometer completes the acquisition of a spectrum. See FAQ: Why doesn’t getSpectrum() return when I think it should? for a discussion of the timing of the *getSpectrum()* method.

Next, you need to call the function to get the wavelengths that correspond to the pixels in the spectrum.

```
double[] allWaveLengths = ocean.getWavelengths(deviceID,
ref errorCode);
```



The two arrays returned from *getSpectrum()* and *getWavelengths()* can be imported to an external program (Microsoft Excel, for instance) and plotted with wavelengths on the x-axis and spectrum on the y-axis, to visualize the spectrum observed by the spectrometer.

There are a number of convenient calls to get one or more indices within the spectrum one-dimensional array given any particular wavelength(s). For example, asking for the index at 340 nm will give the closest approximation based on the spectrometer used.

```
double wl = 0;
double approximateWavlength = 340;
var p_index = ocean.getIndexAtWavelength(deviceId, ref errorCode, ref wl, approximateWavlength);
```

Now the variable p_index can be used to find the pixel value from the array given by *getSpectrum()* call.
In the case where you want many indexes that are not contiguous you can use:

```
double[] waves = { 389, 420, 600 };
```

```
int wvlen = waves.Length;

int[] inx = ocean.getIndicesAtAnyWavelength(deviceId, ref errorCode, ref waves, wvlen);
for (int i = 0; i < wvlen; i++)
{
     Console.Write("Wavelength Index at: {0}  == ", inx[i]);
      Console.WriteLine("Wavelength Value is: {0}\n", waves[i]);
}
```

If a contiguous range is desired, use the following call:

```
double lo = 400;
       double hi = 413;
       Console.WriteLine("Wavelengths from approx {0} nm to {1} nm.", lo, hi);
       int[] inxR = ocean.getIndicesAtWavelengthRange(deviceId, ref errorCode, ref waves, ref
rangeLength, lo, hi);
            for (int i = 0; i < inxR.length; i++)
              {
                  Console.Write("Range Index at: {0}  == ", inxR[i]);
                    Console.WriteLine("Range Value is: {0}\n", waves[i]);
              }
```

Correct for Detector Nonlinearity

All Ocean Insight spectrometers are calibrated at the factory to maximize accuracy. One of the calibrations performed is to correct for detector nonlinearity.

This calibration consists of eight numbers used as the coefficients of a 7th order polynomial that adjusts for the fact that the CCD detectors don't respond to stimuli photons uniformly as more electrons are drained from the well. In other words, the efficiency of the CCD detectors may be 30% when the well is half full, but may be only 20% when the well is completely drained of electrons.

By "efficiency" we mean the probability that an incoming photon will drain an electron from the CCD well; 100% efficiency means every incoming photon will drain one electron, while 50% efficiency means an incoming photon has only a 50% chance of causing an electron to drain.

Nonlinearity calibration is made by averaging all pixels of the CCD array. Thus, we are assuming that all pixels respond about the same.

To ensure that the nonlinearity correction is applied, use the following call:

```
bool nonLinearityOn = true;
ocean.applyNonLinearityCorrection(deviceID, ref errorCode, nonLinearityOn);
```

Testing the state of the nonlinearity correction can be done by calling

```
bool readNonLinearityState = ocean.getNonLinearityCorrectionUsage(deviceID, ref errorCode);
```

Set Integration Time

Integration time is simply the length of time during which we allow light to pass into the spectrometer's detector. In low-light level situations you may want to lengthen this period to obtain a meaningful spectrum. In high-light level situations you may want to shorten this period to avoid saturation. Saturation occurs when the one or more CCD pixels or wells have been completely drained (or filled on some detector models). When this occurs, additional photons entering the "well" have no effect, and the spectrum becomes increasingly distorted. If you plot a saturated spectrum on a graph, it will appear clipped at the peaks.

Integration time is specified in units of microseconds. See the documentation for your spectrometer to determine the allowable minimum and maximum integration times it will support.

To set integration time to 200 msec (200000 microseconds):

```
uint newIntegrationTime = 200000;
ocean.setIntegrationTimeMicros(deviceId, ref errorCode, newIntegrationTime);
```

To retrieve the integration time:

```
var itime = ocean.getIntegrationTimeMicros(deviceId, ref errorCode);
```

Close Spectrometer

When your application is ready to terminate, call the `closeDevice (int deviceID, int% errorCode)` method.

```
ocean.closeDevice(deviceID, ref errorCode);
```

Advanced Features

All spectrometers support basic functions including setting the integration time and acquiring a spectrum. Some spectrometer models offer additional, advanced features.

A characteristic of these features is that the methods or functions you must call to use them are not provided directly in the NetOceanDirect class. Instead, for each feature, NetOceanDirect provides a pair of related methods you can call to determine if a feature is supported by your spectrometer and methods needed to use or control that feature.

Thus, in the OceanDirect, you will see a number of methods whose names look something like `isFeatureEnabled(xxx)` where "*xxx*" is the name of the feature. These methods return a Boolean true/false to indicate whether that spectrometer supports that feature. It is recommended to call the `isFeatureEnabled(xxx)` method *before* attempting to use a feature. If you attempt to use a feature that is not supported by your spectrometer, nothing will happen. See the examples in this section for the correct syntax to call these functions.

After calling the `isFeatureEnabled(xxx)` method (and assuming it returns a "true" value, errorCode=0), the second step is to call the methods unique to that feature. If the errorCode returned is 5 or 24, the feature is not available on the device.

The following example shows the syntax for testing the availability of a feature (Board Temperature) and calling a function that is a member of that feature.

Board Temperature

Some spectrometers contain one or more temperature sensor chips mounted on the printed circuit board (PCB). Your application can read out this temperature value, in degrees Celsius, by means of this feature. Do not confuse this feature with the Thermoelectric Cooling feature, described later in this chapter. Please refer to the spectrometer's documentation to determine which component temperature is returned - e.g., detector temperature or PCB temperature by each sensor.

First check the `isFeatureEnabled` value.

```
bool temperatureInfo = ocean.isFeatureEnabled(deviceID, OceanDirect.FeatureID.FEATURE_ID_TEMPERATURE,
ref errorCode);
```

This will return a true or false.

Then find out how many temperature sensors are available on the spectrometer.

```
int numberOfTempSensors = ocean.AdvancedFeatures().TemperatureController().getCount(deviceID, ref errorCode);
```

Finally, read the temperature of one of the sensors. The sensors are indexed starting with zero – this example gets the temperature of the 0th sensor,

```
int indexZero = 0;

double currentTemperature = ocean.AdvancedFeatures().TemperatureController().getTemperature(deviceID, ref errorCode, indexZero);
```

Below is a list of the advanced features, but be sure to see the documentation for your spectrometer to determine if it supports that specific feature.

Analog In

AnalogIn provides access to information about the device's input pin voltage settings.

To check if AnalogIn is available on your spectrometer, use feature ID: ANALOG_IN.

Analog Out

AnalogOut provides access to information about the device's output pin voltage settings.

To check if AnalogOut is available on your spectrometer, use feature ID: ANALOG_OUT.

Back to Back

BacktoBack provides access to the device's back-to-back scan buffering functionality.

To check if BacktoBack is available on your spectrometer, use feature ID: BACK_TO_BACK.

Continuous Strobe

ContinuousStrobe provides access to the strobe characteristics (e.g., minimum, maximum, width, and period).

To check if ContinuousStrobe is available on your spectrometer, use feature ID: CONTINUOUS_STROBE.

Data Buffer

DataBuffer provides access to onboard memory, allowing you to retrieve stored spectra.

To check if DataBuffer is available on your spectrometer, use feature ID: DATA_BUFFER.

EEPROM

The EEPROM function provides access to EEPROM storage for wavelength calibration and linearity correction coefficients.

To check if EEPROM is available on your spectrometer, use feature ID: EEPROM.

External Trigger Delay

ExternalTriggerDelay provides access to the device's trigger mode parameters, allowing a delay between the trigger and the reading of the spectra.

To check if External Trigger Delay is available on your spectrometer, use feature ID: ACQUISITION_DELAY.

IPv4

This parameter provides access to the device's IPv4 address functionality, allowing the DHCP client to be turned on or off, and to read, add, or delete IP addresses,

To check if IPv4 is available on your spectrometer, use feature ID: IPV4_ADDRESS.

Lamp Enable

This parameter controls the shutter on a light source. These light sources attach directly to an electrical connector on your spectrometer. The lamp will turn on when set to true and turn off when set to false.

To check if Lamp Enable is available on your spectrometer, use feature ID: LIGHT_SOURCE.

LED Activity

Some spectrometers have indicator lights to show power and data transfer status. This feature allows you to access the state of the LEDs.

To check if LED Activity is available on your spectrometer, use feature ID: LED_ACTIVITY.

Pixel Binning

Pixel binning combines the charge collected by several adjacent pixels on the detector, which reduces noise and improves the signal-to-noise ratio.

To check if Pixel Binning is available on your spectrometer, use feature ID: PIXEL_BINNING.

Raw Bus Access

Raw Bus Access allows you to read and write raw bus data.

To check if Raw Bus Access is available on your spectrometer, use feature ID: RAW_BUS_ACCESS.

Scans to Average

Scans to average is another method used to perform noise-reduction (smoothing) on the spectra returned by a spectrometer. With this technique, multiple sequential spectra are averaged to produce a single averaged spectrum. The algorithm uses corresponding pixels from each spectrum when computing the average for a given pixel value. For example, if Scans to Average is set to 5, the pixel[0] values from each of five consecutive scans are added together, and then divided by 5. The resulting value will be reported in pixel[0] of the spectrum returned to the user. This procedure is repeated for each pixel in the spectrum.

NOTE

In the following function call, if you specify a value of 1 for the *setScansToAverage()* argument, no smoothing will be performed. Each spectrum will be reported as is, without any averaging.

Use this call to set the number of scans to average equal to 5 and check the number of spectra averaged:

```
ushort newScansToAverage = 5;
ocean.AdvancedFeatures().SpectrumProcessingController().setScansToAverage(deviceID, ref errorCode,
newScansToAverage);
ushort readScansToAverage =
ocean.AdvancedFeatures().SpectrumProcessingController().getScansToAverage(deviceID, ref errorCode);
```

Spectrum Processing

SpectrumProcessing provides access to on-device averaging, filtering, and triggering functionality.

To check if Spectrum Processing is available on your spectrometer, use feature ID: SPECTRUM_PROCESSING.

NOTE

Triggering functionality will move to SpectrometerProcess in a future release of OceanDirect.

Thermoelectric Cooling

Some spectrometers such as the QE *Pro* have a thermoelectrically cooled (TEC) CCD that can be controlled by the software.

Be careful to observe the allowable temperature range supported by your spectrometer's TE cooler. For example, the TE cooler of the QE *Pro* is capable of dropping the temperature of the CCD by 30-43 degrees Celsius below the ambient temperature. Thus, if the ambient temperature happens to be 25 degrees Celsius, the range of values you may pass in to the *setDetectorSetPointCelsius()* method will be +5 to -18 degrees Celsius. In this example, if you (erroneously) attempt to specify a value of -3 degrees Celsius, the TE cooler will not function properly and the temperature of the CCD will not approach -3 degrees Celsius.

To check if Thermoelectric Cooling is available on your spectrometer, use feature ID: THERMOELECTRIC.

External Trigger Modes

The trigger mode setting of your spectrometer gives you more precise control over the timing of a spectrum capture.

Not all spectrometers support all trigger modes, so be sure to see the documentation for your particular spectrometer. You set the trigger mode of your spectrometer by using the advanced feature `SpectrumProcess` and specifying an integer corresponding to the desired mode as described below.

```
ocean.AdvancedFeatures().SpectrumProcessingController().setTriggerMode(deviceID, ref errorCode, mode)
```

Mode 0: Normal Mode

Sometimes called "free running" mode, this is the default mode for all spectrometers. In this mode, the spectrometer is continuously acquiring new spectra, using default power-up settings for integration time, etc.

The integration time is controlled by calls to the `setIntegrationTimeMicros()` method.

When you call the `getSpectrum()` method, it returns the next available spectrum, subject to delay due to the length of integration time, number of scans to average, and possibly a stability scan. It is also possible that `getSpectrum()` returns almost immediately, even if you specified a lengthy integration period, because the spectrometer is continuously acquiring spectra. See **Appendix A: *FAQs*** for more information.

The `getSpectrum()` method will block (i.e., not return) until the spectrometer finishes acquiring the current spectrum.

Mode 1: External Software Trigger Mode

In this mode, the trigger signal acts like an "enable." As long as the input trigger signal pin on your spectrometer is held electrically high, spectra will be continuously acquired. When the signal goes low, the spectrometer no longer acquires spectra, and *getSpectrum()* will remain blocked (i.e., not return) until the signal goes high again.

This mode is similar to Mode 0 (Normal mode) in that as long as the trigger signal remains high, the spectrometer is continuously acquiring new spectra, irrespective of when you call *getSpectrum()*. In this situation, *getSpectrum()* simply returns the next available spectrum.

The integration time is controlled by calls to the *spectrometerSetIntegrationTimeMicros()* method.

Mode 2: External Synchronization Trigger Mode

In this mode, spectra acquisition is initiated by an external synchronizing TTL trigger signal. The purpose of this mode is to allow multiple spectrometers to be synchronized in terms of the start of their acquisition period, and the duration of the integration period. The integration time is determined based on an average of the length of time between the input trigger signal pulses. See your spectrometer's documentation for the allowed frequency range for this input signal.

Mode 3: Hardware Trigger Mode

In this mode, the spectrometer does not begin to acquire a new spectrum until the rising edge of an external TTL input signal. When you call *getSpectrum()*, this method blocks (i.e., does not return to the caller) until the trigger signal occurs and the spectrum has been acquired.

The integration time is controlled by calling the *spectrometerSetIntegrationTimeMicros()* method.

NOTE

Once you put the spectrometer in this mode, it is impossible to programmatically return to normal Hardware Trigger mode. You must power-cycle the spectrometer by unplugging it from your computer.

Mode 4: Single-Shot Trigger Mode (Quasi Realtime Mode)

This mode is designed to provide more precise software control over when a spectrum acquisition is initiated. This is especially valuable when you need to use very long integration periods, but also want precise control over when the acquisition period begins.

This mode is only available on certain spectrometers.

In this mode, the spectrometer automatically sets its integration time to a very short integration period and then begins to continuously acquire spectra using this minimal integration time. When *spectrometerSetIntegrationTimeMicros()* is called to specify the desired integration period, this value is stored in the spectrometer, but the spectrometer continues to acquire spectra using the minimal integration period.

When *getSpectrum()* is called, the spectrometer completes the current acquisition (which should happen very quickly since the integration time is so short). The spectrometer then sets its integration time to the value requested and initiates a new spectrum acquisition. When this acquisition completes, *getSpectrum()* returns the new spectrum. The spectrometer once again reverts to the minimal integration period and continues to acquire spectra in the background.

Developing Your Application

You may develop your application in any environment of your choosing. We will focus on Microsoft Visual Studio examples due to its availability and cross-platform capabilities.

Microsoft Visual Studio

To use the .NET assembly interface you must add a reference to the assembly as follows:

1. In your application, click on the **Project** menu item, and then choose **Add Reference**.
2. Click on the **Browse** tab.
3. Navigate to the OCEANDIRECT_HOME directory.
4. Highlight **NetOceanDirect.dll** and click **OK**.

Creating a New C# Project That Uses the .NET Interface to OceanDirect

1. Create new project of type C# “Windows Console”.
2. Add a reference to netOceanDirect.dll:
    1. In Solution Explorer, right-click on **References** and choose **Add Reference…**
    2. Click the **Browse** tab.
    3. Navigate to the OCEANDIRECT_HOME folder.
    4. Highlight **NetOceanDirect.dll** and click **OK**.

Deploying Your C# Application

Normally, all that is needed to deploy your C# application is the EXE file containing the application itself. And you will also need to deploy the appropriate OceanDirect “redistributable” installer.

However, if your C# application uses the OceanDirect .NET assembly interface, you must also ensure that a copy of NETOceanDirect.dll is placed in the same folder as your application's EXE file. You can obtain the DLL file from the OCEANDIRECT_HOME directory.

LabVIEW

OceanDirect provides a .NET 4.0 interface. We recommend that all LabVIEW applications use the .NET 4.0 interface when accessing OceanDirect functions.

LabVIEW is able to show all the methods in a class as well as each method's inputs and outputs with default named variables. This graphical representation clearly documents the .NET interface within the LabVIEW environment.

The following steps must be performed when starting a new LabVIEW projects:

1. The first step is to place an "Invoke Node (.NET)" on the Block Diagram panel. Do this in the Functions window by selecting **Connectivity -> .NET -> Invoke Node (.NET)** and dragging it to the panel. Right-click the node and then choose **Select Class -> .NET -> Browse**. Navigate to the NetOceanDirect.dll. The default location for the DLL is C:\Program Files\Ocean Insight\OceanDirect SDK\lib\.
2. In the **Select Object From Assembly** window, select the **OceanDirect** object on the panel, click **OK**. This updates the node on the block diagram to "OceanDirect". Right-click on the node and choose **Select Method.** A list of all available methods is displayed. Select **[S]getInstance**. This returns a .NET object, which will be your primary OceanDirect object that you call all further methods from.
3. Create another Invoke Node of class `OceanDirect` and select the method **findDevices**. This method returns an array of devices that OceanDirect knows how to communicate with. You must subset the array and call the Property Node (ID) to find each device's assigned **deviceID**. The **deviceID** will be used for all function calls that perform an action on a specific device.
4. You must then call **openDevice** from the OceanDirect .NET object that was returned earlier from **getInstance**. The **openDevice** method requires you to pass in the **deviceID** and a number as a reference (LabVIEW handles the reference part automatically). It will return an **errorCode,** which you should verify is 0 (0=no error) before performing your next function.
5. Once the device is open you can call any of the standard methods found in the returned instance, e.g., **getWavelength**, **getSpectrum**,**setIntegrationTimeMicro**, etc.

Visual Basic

Using the .NET Interface to OceanDirect

To use the .NET assembly interface to OceanDirect in your Visual Basic application, you must add a reference to your assembly as follows:

1. Click on the **Project** menu item and choose **Add Reference**.
2. Click on the **Browse** tab.
3. Navigate to the **OCEANDIRECT_HOME** directory.
4. Highlight the **netOceanDirect.dll** and click **OK**.

Sample Programs

A collection of sample programs for OceanDirect demonstrating basic functionality may be downloaded from the OceanDirect product page at OceanInsight.com.

Appendix A - Data Structures

The OceanDirect API is the collection of objects and methods your application uses to control spectrometers and acquire data from them.

Depending on your development environment, you will use OcearDirect.dll or netOceanDirect.dll (used in the Visual Studio environment). Public member functions for each class of the NetOceanDirect data structures are shown below. OceanDirect has similar names for the public functions.

The table below is a quick reference showing each class and the public member functions. The items in the table link to a description of the functions in this document. A more detailed description of the functions and the associated parameters may be found in the reference manuals “OceanDirect.rtf” located at “c:\Program Files\Ocean Insight\OceanDirect SDK\Doc\rtf\OceanDirectWrapper_User_Manual.rtf”, and “NetOceanDirect User Manual.rtf” located at “c:\Program Files\Ocean Insight\OceanDirect SDK\Doc\net\rtf\ NetOceanDirect_User_Manual.rtf”.

HTML versions of the reference manual may be found at “c:\Program Files\Ocean Insight\OceanDirect SDK\Doc\html\index.html\” and “c:\Program Files\Ocean Insight\OceanDirect SDK\Doc\net\index.html\”.

Public Functions

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
AcquireDelay	oceandirect::api::AcquireDelayAPI	NetOceanDirect::AcquireDelay	getDelayIncrementMicroseconds
			getDelayMaximumMicroseconds
			getDelayMicroseconds
			getDelayMinimumMicroseconds
			setDelayMicroseconds
Advanced	oceandirect::api::Advance	NetOceanDirect::Advanced	AcquireDelayController
			AnalogInController
			AnalogOutController

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
			BackToBackController
			ContinuousStrobeController
			DataBufferController
			DeviceRevisionController
			EepromController
			GpioController
			Ipv4Controller
			IrradianceController
			LampController
			LedActivityController
			LightSourceController
			NonLinearityController
			OpticalBenchController
			RawBusController
			SingleStrobeController
			SpectrumProcessingController
			TECController
			TemperatureController
Analogin	oceandirect::api::AnalogInAPI	NetOceanDirect::AnalogIn	configureAltPin
			getMaximumInputVoltageDC
			getMinimumInputVoltageDC
			getNumberOfVoltageDCInputs
			getVoltageDCInputVolts

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
AnalogOut	oceandirect::api::AnalogOutAPI	NetOceanDirect::AnalogOut	configureAltPin
			getMaximumOutputVoltageDC
			getMinimumOutputVoltageDC
			getNumberOfVoltageDCOutputs
			setVoltageDCOutVolts
BackToBack	oceandirect::api::BackToBackAPI	NetOceanDirect::BackToBack	getNumberOfBackToBackScans
			setNumberOfBackToBackScans
ContinuousStrobe	oceandirect::api::ContinuousStrobeAPI	NetOceanDirect::ContinuousStrobe	getContinuousStrobeEnable
			getContinuousStrobePeriodIncrementMicroseconds
			getContinuousStrobePeriodMaximumMicroseconds
			getContinuousStrobePeriodMicroseconds
			getContinuousStrobePeriodMinimumMicroseconds
			setContinuousStrobeEnable
			setContinuousStrobePeriodMicroseconds
			setContinuousStrobeWidthMicroseconds
DataBuffer	oceandirect::api::DataBufferAPI	NetOceanDirect::DataBuffer	clear
			getBufferCapacity
			getBufferCapacityMaximum
			getBufferCapacityMinimum
			getBufferEnable

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
			getNumberOfElements
			setBufferCapacity
			setBufferEnable
DeviceRevision	oceandirect::api::DeviceRevisionAPI	NetOceanDirect::DeviceRevision	getRevisionFirmware
			getRevisionHardware
Eeprom	oceandirect::api::EepromAPI	NetOceanDirect::Eeprom	readSlot
Gpio	oceandirect::api::GpioAPI	NetOceanDirect::Gpio	getNumberOfGPIO
			getOutputEnable (specific pin)
			getOutputEnable (all pins)
			getValue(specific pin)
			getValue(all pins)
			setOutputEnable (specific pin)
			setOutputEnable (all pins)
			setValue (specific pin)
			setValue (all pins)
Ipv4Address	oceandirect::Ipv4AddressAPI	NetOceanDirect::Ipv4Address	isDHCPEnabled
			setDHCPEnable
			getNumberOffIpAddresses
			readIpAddress
			addStaticIpAddress

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
			deleteStaticIpAddress
IrradCalibrate	oceandirect::api::IrradCalibrAPI	NetOceanDirect::IrradCalibrate	read
			readCalibrationDataSize
			readCollectionArea
			write
			writeCollectionArea
Lamp	oceandirect::api::LampAPI	NetOceanDirect::Lamp	setEnable
LedActivity	oceandirect::api::LedActivityAPI	NetOceanDirect::LedActivity	isEnabled
			setEnable
LightSource	oceandirect::api::LightSourceAPI	NetOceanDirect::LightSource	getCount
			getIntensity
			hasEnable
			hasVariableIntensity
			isEnabled
			setEnable
			setIntensity
			setLampEnable
NonLinearity	oceandirect::api::NonLinearityAPI	NetOceanDirect::NonLinearity	getCoeffs

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
OceanDirect	oceandirect::api::OceanDirectAPI	NetOceanDirect::OceanDirect	AdvancedFeatures
			applyElectricDarkCorrection
			applyNonLinearityCorrection
			closeDevice
			findDevices
			getApiVersion
			getDeviceModel
			getCurrentDevicesConnected
			getEDPCount
			getEDPIndices
			getElectricDarkCorrectionUsage
			getIndexAtWavelength
			getIndicesAtAnyWavelength
			getIndicesAtWavelengthRange
			getIntegrationTimeMicros
			getMaximumIntegrationTime
			getMaximumIntensity
			getMinimumIntegrationTime
			getNonLinearityCorrectionUsage
			getNumberOfPixels
			getRawSpectrumWithMetadata
			getSerialNumber
			getSpectrum
			getWavelength

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
			getWavelengths
			isFeatureEnabled
			openDevice
			setIntegrationTimeMicros
OpticalBench	oceandirect::api::OpticalBenchAPI	NetOceanDirect::OpticalBench	getSlitWidthMicrons
			getSerialNumber
			getCoating
			getArrayWavelength
			getLensInstalled
			getFilter
			getGrating
			getSerialNumber
RawBus	oceandirect::api::RawBusAPI	NetOceanDirect::RawBus	accessUsbRead
			accessEthRead
			accessEthWrite
			accessUsbWrite
			getStringDescriptor
SingleStrobe	oceandirect::api::SingleStrobeAPI	NetOceanDirect::SingleStrobe	getStrobeDelay
			getStrobeEnable
			getStrobeIncrementDelay
			getStrobeIncrementWidth

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
			getStrobeMaximumCycle
			getStrobeMaximumDelay
			getStrobeMaximumWidth
			getStrobeMinimumDelay
			getStrobeMinimumWidth
			getStrobeWidth
			setStrobeDelay
			setStrobeEnable
			setStrobeWidth
SpectrumProcess	oceandirect::api::SpectrumProcessAPI	NetOceanDirect::SpectrumProcess	getBoxcarWidth
			getScansToAverage
			setBoxcarWidth
			setScansToAverage
			setTriggerMode
SpectrumWithMetadata		NetOceanDirect::SpectrumWithMetadata	SpectrumWithMetadata
Temperature	oceandirect::api::TemperatureAPI	NetOceanDirect::Temperature	getCount
			getTemperature
ThermoElectric	oceandirect::api::ThermoElectricAPI	NetOceanDirect::ThermoElectric	getEnable
			getFanEnable
			getSetpoint

General Class	OceanDirect Class	NetOceanDirect Class	C# Public Functions
			getStable
			readTemperatureDegreesC
			setEnable
			setFanEnable
			setTemperatureSetpointDegreesC

AcquireDelay

void ***setDelayMicroseconds*** (long deviceID, int% errorCode, unsigned long delay_usec)

- Sets acquisition delay (time between when device receives trigger signal and when integration begins) for the given device. Functions such as ***getDelayMaximumMicroseconds()*** can be used to determine valid values for this setting.

unsigned long ***getDelayMicroseconds*** (long deviceID, int% errorCode)

- Retrieves the given device's current acquisition delay setting (time between when device receives trigger signal and when integration begins).

unsigned long ***getDelayIncrementMicroseconds*** (long deviceID, int% errorCode)

- Retrieves the given device's acquisition delay resolution, which is the minimum delta between possible consecutive acquisition delay settings. The device supports a range of acquisition delay settings [min, max] with intermediate values spaced apart by this quantity, which forms the following range in MATLAB notation: ***getDelayMinimumMicroseconds() : getDelayIncrementMicroseconds() : getDelayMaximumMicroseconds()***

unsigned long ***getDelayMaximumMicroseconds*** (long deviceID, int% errorCode)

- Retrieves the given device's maximum possible acquisition delay setting.

unsigned long ***getDelayMinimumMicroseconds*** (long deviceID, int% errorCode)

- Retrieves the given device's minimum possible acquisition delay setting.

Advanced

DeviceRevision ***DeviceRevisionController*** ()

- Returns a tracked handle to an instance of the SerialNumber feature, which provides access to a device's serial number.

ContinuousStrobe ***ContinuousStrobeController*** ()

- Returns a tracked handle to an instance of the continuous strobe controller, which provides access to the device's continuous strobe functionality.

SingleStrobe ***SingleStrobeController*** ()

- Returns a tracked handle to an instance of the single strobe controller, which provides access to the device's single-strobe functionality.

ThermoElectric ***TECController*** ()

- Returns a tracked handle to an instance of the TEC controller, which provides access to the device's Thermoelectric Cooler (TEC) functionality.

AnalogIn ***AnalogInController*** ()

- Returns a tracked handle to an instance of the analog in controller, which provides access to information about the device's input pin voltage settings.

AnalogOut ***AnalogOutController*** ()

- Returns a tracked handle to an instance of the analog out controller, which provides access to information about the device's output pin voltage settings.

IrradCalibrate ***IrradianceController*** ()

- Returns a tracked handle to an instance of the irradiance calibration controller, which provides access to the device's onboard irradiance calibration.

EEPROM ***EepromController*** ()

- Returns a tracked handle to an instance of the EEPROM controller, which provides access to the device's onboard Electrically-Erasable ReProgrammable Memory (EEPROM), which is used to store device-specific information in nonvolatile fashion.

Lamp ***LampController*** ()

- Returns a tracked handle to an instance of the lamp enable controller, which provides access to toggle the device's lamp enable.

LedActivity ***LedActivityController*** ()

- Returns a tracked handle to an instance of the LED status controller, which provides access to the device's outside LED control interface.

LightSource ***LightSourceController*** ()

- Returns a tracked handle to an instance of the light source controller, which provides access to the device's light-source control interface.

SpectrumProcess ***SpectrumProcessingController*** ()

- Returns a tracked handle to an instance of the spectrum processing controller, which provides access to on-device averaging, filtering, and triggering functionality.

Temperature ***TemperatureController*** ()

- Returns a tracked handle to an instance of the temperature controller, which provides access to the device's onboard temperature sensors (but not its TEC, which, if applicable, is controlled via TECController())

DataBuffer ***DataBufferController*** ()

- Returns a tracked handle to an instance of the data buffer controller, which provides access to the device's onboard data buffering functionality, which allows it to store spectral measurements onboard for later retrieval.

BackToBack ***BackToBackController*** ()

- Returns a tracked handle to an instance of the back-to-back scans controller, which provides access to the device's back-to-back scan buffering functionality, which allows it to collect multiple scans immediately following one another (back-to-back) for maximum acquisition rate.

OpticalBench ***OpticalBenchController*** ()

- Returns a tracked handle to an instance of the optical bench controller, which provides access to information about the device's optical bench, such as its slit width.

AcquireDelay ***AcquireDelayController*** ()

- Returns a tracked handle to an instance of the acquisition delay controller, which provides access to information about the device's acquisition delay settings, which govern the delay between the receipt of a trigger signal and the actual start of integration.

RawBus ***RawBusController*** ()

- Returns a tracked handle to an instance of the raw BUS controller, which provides read/write access to the selected BUS port used to communicate with the device.

NonLinearity ***NonLinearityController*** ()

- Returns a tracked handle to an instance of the nonlinearity correction controller, which provides access to the nonlinearity correction model used to correct spectral measurements for detector nonlinearity.

Gpio ***GpioController*** ()

- Returns a tracked handle to an instance of the GPIO controller, which provides access to the device's General-Purpose Input/Output pins.

Ipv4Address ***Ipv4Controller*** ()

- Returns a tracked handle to an instance of the IPv4 controller, which provides access to the device's IPv4 address functionality.

AnalogIn

int ***getNumberOfVoltageDCInputs*** (long deviceID, int% errorCode)

- Gets the total number of voltage inputs that are supported by this feature.

double ***getMaximumInputVoltageDC*** (long deviceID, int% errorCode)

-
- Get the most positive voltage that can be sampled by the indicated input.

double ***getMinimumInputVoltageDC*** (long deviceID, int% errorCode)

- Get the most negative voltage that can be sampled by the indicated input.

double ***getVoltageDCInputVolts*** (long deviceID, int% errorCode, int input)

- Get the latest measured voltage from the given input. Assuming that there may be multiple inputs on the device, the index of the input must be provided. These will start at zero and increase from there.

void ***configureAltPin*** (long deviceID, int% errorCode, int pinNumber, int pinType)

- Sets a particular pin to a particular usage type.

AnalogOut

int ***getNumberOfVoltageDCOutputs*** (long deviceID, int% errorCode)

- Gets the number of voltage controlled outputs available in this feature.

double ***getMaximumOutputVoltageDC*** (long deviceID, int% errorCode)

- Gets the most positive voltage that this feature can produce.

double ***getMinimumOutputVoltageDC*** (long deviceID, int% errorCode)

- Gets the most negative voltage that this feature can produce.

void ***setVoltageDCOutVolts*** (long deviceID, int% errorCode, int output, double voltage)

- Sets the voltage to be driven on the indicated output.

void ***configureAltPin*** (long deviceID, int% errorCode, int pinNumber, int pinType)

- Sets a particular pin to a particular usage type.

BackToBack

unsigned long ***getNumberOfBackToBackScans*** (long deviceID, int% errorCode)

- Retrieves the given device's current back-to-back-scans configuration.

void ***setNumberOfBackToBackScans*** (long deviceID, int% errorCode, unsigned long numScans)

- Sets the given device's back-to-back-scans configuration to the given value.

void ***setDefaultFactor2*** (long deviceID, int% errorCode, const unsigned char binningFactor)

- Changes the default value for the given device's pixel binning factor setting to a new value. This new default will then be applied each time the device powers on, or when ***setDefaultFactor()*** is called.

unsigned char ***getDefaultFactor*** (long deviceID, int% errorCode)

- Retrieves the given device's default pixel binning factor setting.

unsigned char ***getMaxFactor*** (long deviceID, int% errorCode)

- Retrieves the given device's maximum possible pixel binning factor setting.

ContinuousStrobe

unsigned long ***getContinuousStrobePeriodMicroseconds*** (long deviceID, int %errorCode)

- Gets the given device's continuous strobe period, which is the amount of time required for one complete strobe cycle (low and high).

bool ***getContinuousStrobeEnable*** (long deviceID, int %errorCode)

- Determines if the continuous strobe is enabled or disabled.

unsigned long ***getContinuousStrobePeriodMinimumMicroseconds*** (long deviceID, int %errorCode)

- Gets the minimum continuous strobe period in microseconds.

unsigned long ***getContinuousStrobePeriodMaximumMicroseconds*** (long deviceID, int %errorCode)

- Gets the maximum continuous strobe period in microseconds.

unsigned long ***getContinuousStrobePeriodIncrementMicroseconds*** (long deviceID, int %errorCode)

- Retrieves the given device's continuous strobe period resolution in microseconds.

unsigned long ***getContinuousStrobeWidthMicroseconds*** (long deviceID, int %errorCode)

- Gets the continuous strobe width in microseconds.

void ***setContinuousStrobeEnable*** (long deviceID, int %errorCode, bool strobeEnable)

- Enables/disables the continuous strobe output of the given device.

void ***setContinuousStrobePeriodMicroseconds*** (long deviceID, int %errorCode, unsigned long strobePeriodMicroseconds)

- Sets the continuous strobe period in microseconds.

void ***setContinuousStrobeWidthMicroseconds*** (long deviceID, int %errorCode, unsigned long strobeWidthMicroseconds)

- Sets the continuous strobe width in microseconds.

DataBuffer

void ***clear*** (long deviceID, int% errorCode)

- Clears the given device's onboard data buffer by removing all buffered measurements. After this operation, getNumberOfElements() will return 0.

unsigned long ***getNumberOfElements*** (long deviceID, int% errorCode)

- Retrieves the number of elements (spectra) currently present in the given device's onboard data buffer.

unsigned long ***getBufferCapacity*** (long deviceID, int% errorCode)

- Retrieves the given device's current onboard data buffer capacity, i.e., the maximum number of spectra it can store before the buffer is filled.

unsigned long ***getBufferCapacityMaximum*** (long deviceID, int% errorCode)

- Retrieves the given device's maximum possible onboard data buffer capacity setting.

unsigned long ***getBufferCapacityMinimum*** (long deviceID, int% errorCode)

- Retrieves the given device's minimum possible onboard data buffer capacity setting.

void ***setBufferCapacity*** (long deviceID, int% errorCode, unsigned long capacity)

- Sets the given device's onboard data buffer capacity, which governs how many spectra it will store at once.

void ***setBufferEnable*** (long deviceID, int% errorCode, bool enabled)

- Enables/disables the data buffering feature of the given device.

bool ***getBufferEnable*** (long deviceID, int% errorCode)

- Checks to see whether data buffering is currently enabled on the given device.

DeviceRevision

string ***getRevisionHardware*** (long deviceID, int% errorCode)

- Retrieves the hardware revision of the given device or sets an error code if this is not available.

string int ***getRevisionFirmware*** (long deviceID, int% errorCode)

- Retrieves the firmware revision of the given device or sets an error code if this is not available.

Devices

Data Fields

- property int ***Id***
- property String ***Name***
- property bool ***InUse***
- property BusType ***Btype***
- property int ***SpectrumLen***

Eeprom

array< unsigned char > ***readSlot*** (long deviceID, int% errorCode, int slotNumber, int bufferLength)

- Reads a specific slot from the given device's EEPROM and returns its contents in a new tracked array.

GPIO

int ***getNumberOfGPIO*** (long deviceID, int% errorCode)

- Retrieves the number of GPIO pins available on the given device

void ***setOutputEnable*** (long deviceID, int% errorCode, int bit, bool direction)

- Configures a specific GPIO pin as either an input or an output on a given device.

void ***setOutputEnable*** (long deviceID, int% errorCode, uint32_t bitmask)

- Configures the entire bank of GPIO pins as inputs or outputs using a bitmask.

bool ***getOutputEnable*** (long deviceID, int% errorCode, int bit)

- Retrieves the current direction (input vs. output) for a specific GPIO pin on the given device.

uint32_t ***getOutputEnable*** (long deviceID, int% errorCode)

- Retrieves the current direction of all GPIO pins on the given device.

void ***setValue*** (long deviceID, int% errorCode, int bit, bool value)

- Sets the value of a specific GPIO pins on the given device.

void ***setValue*** (long deviceID, int% errorCode, uint32_t bitmask)

- Sets the value of all GPIO pins on the given device.

bool ***getValue*** (long deviceID, int% errorCode, int bit)

- Retrieves the current logic level of a specific GPIO pin on the given device.

uint32_t ***getValue*** (long deviceID, int% errorCode)

- Retrieves the current logic level of all GPIO pins on the given device.

Ipv4

bool ***isDHCPEnabled*** (long deviceID, int% errorCode, unsigned char ifNum)

- Check to see if DHCP (client) is enabled on the specified interface. If DHCP is enabled then the device will be able to receive and IP address from a DHCP server oin the network it is connected to.

void ***setDHCPEnable*** (long deviceID, int %errorCode, unsigned char ifNum, unsigned char enabled)

- Turn the DHCP client on or off for the device on the specified interface.

int ***getNumberOfIpAddresses*** (long deviceID, int% errorCode, unsigned char ifNum)

- Get the number of IP addresses available on the specified interface. If DHCP is enabled on the specified interface then index 0 represents the DHCP address and the following addresses will be any static IP addresses.

array< unsigned char > ***readIpAddress*** (long deviceID, int% errorCode, unsigned char ifNum, unsigned char addressIndex, unsigned int% netmask)

- Retrieve the IP address and netmask on the specified interface. If DHCP is enabled on the specified interface then index 0 represents the DHCP address and the following addresses will be any static IP addresses. The IP address is returned as 4 bytes into a user supplied array. The leading part of the IP address will be in the first element of the array, followed by the remaining parts in order to the last part of the IP address in the fourth element of the array.

void ***addStaticIpAddress*** (long deviceID, int% errorCode, unsigned char ifNum, array< unsigned char >^% ipAddress, unsigned int netmask)

- Add a static IP address to the specified interface. The IP address is specified as 4 bytes in an array. The leading part of the IP address must contain the first element of the array, followed by the remaining parts in order to the last part of the IP address in the fourth element of the array.

void ***deleteStaticIpAddress*** (long deviceID, int% errorCode, unsigned char ifNum, unsigned char addressIndex)

- Delete a static IP address on the specified interface.

IrradCalibrate

array< double > ***read*** (long deviceID, int% errorCode, int bufferLength)

- Reads the given device's onboard irradiance calibration. The buffer length can be computed using the return values of readCalibrationDataSize().

int ***readCalibrationDataSize*** (long deviceID, int% errorCode)

- Reads the given device's onboard irradiance calibration data count.

int ***write*** (long deviceID, int% errorCode, array< float >^ buffer, int bufferLength)

- Writes an irradiance calibration to the given device's onboard memory.

float ***readCollectionArea*** (long deviceID, int% errorCode)

- Retrieves the amount of area used to collect light for the given device.

void ***writeCollectionArea*** (long deviceID, int% errorCode, float area)

- Sets the amount of area used to collect light (as an abstract value used for irradiance calculations, not by actually changing anything on the detector) for the given device.

Lamp

void ***Lamp::setEnable*** (long deviceID, int% errorCode, bool enable)

- Enables/disables the specified strobe lamp connected to the given device.

LedActivity

bool ***LedActivity::isEnabled*** (long deviceID, int% errorCode)

- Retrieves whether the LED activity light connected to the given device is currently enabled.

void ***LedActivity::setEnable*** (long deviceID, int% errorCode, bool enable)

- Enables/disables the specified light source connected to the given device.

LightSource

int ***getCount*** (long deviceID, int% errorCode)

- Gets the number of light sources that are connected to the given device. Such light sources could be individual LEDs, light bulbs, lasers, etc. Each of these light sources may have different capabilities, such as programmable intensities and enable controls, which should be queried before they are used.

bool ***hasEnable*** (long deviceID, int% errorCode, int lightSourceIndex)

- Queries whether the indicated light source connected to the given device has a usable enable/disable control. If this returns 0 (meaning no enable available) then calling isEnabled() or setEnable() will fail.

bool ***isEnabled*** (long deviceID, int% errorCode, int lightSourceIndex)

- Retrieves whether the specified light source connected to the given device is currently enabled. This function should not be called if hasEnable() returns false for this light source.

void ***setEnable*** (long deviceID, int% errorCode, int lightSourceIndex, bool enable)

- Enables/disables the specified light source connected to the given device.

bool ***hasVariableIntensity*** (long deviceID, int% errorCode, int lightSourceIndex)

- Queries the given device to check whether the specified connected light source supports variable intensity.

double ***getIntensity*** (long deviceID, int% errorCode, int lightSourceIndex)

- Retrieves the current intensity setting of a light source connected to the given device. The light source specified must support variable intensity for this to work. The intensity is normalized over the range [0, 1], with 0 as the minimum and 1 as the maximum.

void ***setIntensity*** (long deviceID, int% errorCode, int lightSourceIndex, double intensity)

- Sets the intensity of a light source connected to the given device. The light source specified must support variable intensity for this to work. The intensity is normalized over the range [0, 1], with 0 as the minimum and 1 as the maximum.

  SAFETY WARNING: A light source at its minimum intensity (0) might still emit light, and in some cases, this light may be harmful radiation. A value of 0 indicates the minimum of the programmable range for the light source, and does not necessarily turn the light source off. To disable a light source completely, use setEnable() if the device supports this feature, or provide some other mechanism to allow the light to be disabled or blocked by the operator.

  In some cases, the intensity may refer to the duty cycle of a pulsed light source instead of a continuous power rating. The actual power output of the light source might not vary linearly with the reported intensity, so independent measurement or calibration of the light source may be necessary.

int ***setLampEnable*** (long deviceID, int% errorCode, bool state)

- Sets the strobe enable state of the given device.

NonLinearity

array< double > getCoeffs (long deviceID, int% errorCode, int length)

- Retrieve the nonlinearity correction coefficients from the given device

OceanDirect

OceanDirect ()

Advanced AdvancedFeatures ()

- Retrieves a tracked handle to an Advanced instance, which may be used to access advanced hardware features beyond what is accessible from the OceanDirect class alone, such as trigger delay or raw USB access.

void ***openDevice*** (int deviceID, int% errorCode)

- Opens the spectrometer with the given device ID, which allows the various operations to be performed on it by passing this same device ID. After being opened, the spectrometer will not be opened again until it is first closed using closeDevice().

void ***closeDevice*** (int deviceID, int% errorCode)

- Closes the spectrometer with the given ID, which cleans up associated resources and cached values and allows it to be disconnected from the computer with no negative repurcussions. Only spectrometers that have already been opened using openDevice() should be closed.

array< Devices^> ***findDevices*** ()

- Finds all available Ocean devices by scanning on USB for devices with Ocean drivers, finding devices that respond to UDP multicast (FX and HDX), and returning IDs for any TCP-enabled devices that have been manually specified. Returns a list of Devices objects corresponding to all of these devices, which contain some associated metadata, including their device IDs, which may be used to refer to them using the other OceanDirect member functions (starting with openDevice()).

array< *Devices^*> ***getCurrentDevicesConnected*** *()*

- Finds all available Ocean devices that are currently attached by scanning on USB for devices with Ocean drivers. Returns information whether device is in use or not.

string ***getDeviceModel*** *(long deviceID, int% errorCode)*

- Retrieves the model name from the specified device.

double ***getApiVersion*** ()

- Retrieves the API version number for the Ocean Direct SDK.

bool ***isFeatureEnabled*** (int deviceID, NetOceanDirect::OceanDirect::FeatureID featureId, int% errorCode)

- Determine if a specified feature is supported by a spectrometer or not.

void ***applyElectricDarkCorrection***(int deviceID, int% errorCode, bool apply)

- Sets whether electric dark correction is used.

bool ***getElectricDarkCorrectionUsage***(int deviceID, int% errorCode)

- Gets electric dark correction useage from a device.

void ***applyNonLinearityCorrection***(int deviceID, int% errorCode, bool apply)

- Sets whether nonlineaity correction is used.

bool ***getNonLinearityCorrectionUsage***(int deviceID, int% errorCode)

- Gets nonlinearity correction usage from the given device.

void ***setIntegrationTimeMicros*** (int deviceID, int% errorCode, unsigned long integrationTimeMicros)

- Sets integration time for the given device, which is how long the detector collects photons before reading it out.

unsigned long ***getIntegrationTimeMicros*** (int deviceID, int% errorCode)

- Gets current integration time from the given device.

array< double > ***getWavelengths*** (int deviceID, int% errorCode)

- Gets wavelength at which each pixel on the device's detector reports intensity.

double ***getWavelength*** (int deviceID, int% errorCode, int pixel)

- Gets wavelength for a single pixel on the detector at the specified index.

array< double > ***getSpectrum*** (int deviceID, int% errorCode)

- Retrieves a spectral measurement in a manner depending on the device's current trigger mode.

array< ***SpectrumWithMetadata***^> ***getRawSpectrumWithMetadata*** (int deviceID, int% errorCode)

- Retrieves raw spectrum with timestamp. To use this function, the data buffer must be enabled and the back-to-back spectra acquisition count must be set.

int ***getNumberOfPixels*** (int deviceID, int% errorCode)

- Gets the number of pixels on the detector, which matches the length of spectra returned by getSpectrum or wavelengths returned by getWavelengths(). Depends on device and current pixel binning mode, if applicable. Does not actually communicate with the device, since this value is read during openDevice() and cached in OceanDirect.Pixels (int deviceID, int% errorCode).

int ***getIndexAtWavelength*** (long deviceID, int% errorCode, double% wavelength, double aproxWavelength)

- Determines the pixel index on the given device that measures light at a wavelength closest to the given query wavelength. Does not communicate with the device - uses cached wavelengths.

array< int > ***getIndicesAtAnyWavelength*** (long deviceID, int% errorCode, array< double >^% wavelength, int length)

- As getIndexAtWavelength, but looks up multiple query wavelengths and returns an index for each.

array< int > ***getIndicesAtWavelengthRange*** (long deviceID, int% errorCode, array< double >^% wavelength,, double lo, double hi)

- Retrieves the indices and wavelengths of pixels that measure light at wavelengths within the given spectral range.

int ***getEDPCount*** (int deviceID, int% errorCode)

- Retrieves the number of electric dark pixels on the detector of the given device. Electric dark pixels are optically masked so that they receive no light. Does not communicate with the device - uses cached values.

array< int > ***getEDPIndices*** (int deviceID, int% errorCode, int length)

- Retrieves the indices of electric dark pixels on the detector of the given device. Electric dark pixels are optically masked so that they receive no light. Does not communicate with the device - uses cached values.

double ***getMaximumIntensity*** (int deviceID, int% errorCode)

- Returns the greatest intensity value that could possibly be reported at any pixel by the given device (i.e., by getSpectrum()). Does not communicate with the device - uses cached values.

unsigned long ***getMinimumIntegrationTime*** (int deviceID, int% errorCode)

- Returns the minimum allowed integration time for the given device. Trying to set the device's integration time to a value below this minimum will result in setIntegrationTimeMicros() silently clamping the given time to this minimum value. Does not communicate with the device - uses cached values.

unsigned long ***getMaximumIntegrationTime*** (int deviceID, int% errorCode)

- Returns the maximum allowed integration time for the given device. Trying to set the device's integration time to a value above this maximum will result in setIntegrationTimeMicros() silently clamping the given time to this maximum value. Does not communicate with the device - uses cached values.

string ***getSerialNumber*** (long deviceID, int% errorCode)

- Retrieves the serial number of the spectrometer.

OpticalBench

string ***getArrayWavelength***(long deviceID, int% errorCode)

- Retrieves the optical array wavelength from a given device.

string ***getSlitWidthMicrons*** (long deviceID, int% errorCode)

- Retrieves the slit width from a given device.

string ***getSerialNumber*** (long deviceID, int% errorCode)

- Retrieves the optical bench serial (detector) number from a given device.

string ***getCoating*** (long deviceID, int% errorCode)

- Retrieves the name of the optical bench coating from a given device.

string ***getFilter*** (long deviceID, int% errorCode)

- Retrieves the name of the optical bench filter from a given device.

string ***getGrating*** (long deviceID, int% errorCode)

- Retrieves the name of the optical bench diffraction grating from a given device.

string ***getLensInstalled*** (long deviceID, int% errorCode)

- Retrieves whether a lens is installed for a given device.

string ***getSerialNumber*** (long deviceID, int% errorCode)

- Retrieves the optical bench detector serial number from a given device.

RawBus

array< unsigned char > ***accessUsbRead*** (long deviceID, int% errorCode, int bufferLength, unsigned char endpoint)

- Reads data directly from the USB endpoint through which the given device is attached.

int ***accessUsbWrite*** (long deviceID, int% errorCode, array< unsigned char >^% buffer, int bufferLength, unsigned char endpoint)

- Writes data directly to the USB endpoint through which the given device is attached.

array< unsigned char > ***accessEthRead*** (long deviceID, int% errorCode, int bufferLength)

- Reads data from the specified device over Ethernet

int ***accessEthWrite*** (long deviceID, int% errorCode, array< unsigned char >^% buffer, int bufferLength)

- Writes data to the specified device over Ethernet.

String ***getStringDescriptor*** (long deviceID, int% errorCode, int index)

- Reads the USB device descriptor reported by the given device.

SingleStrobe

void ***setStrobeEnable*** (long deviceID, int% errorCode, bool strobeEnable)

- Sets the enable status of the single strobe signal. Note that on some devices the enable control is shared with other signals (e.g., lamp enable and continuous strobe) so this may have some side-effects and changing those features may affect the single strobe as well.

void ***setStrobeDelay*** (long deviceID, int% errorCode, unsigned long microseconds)

- Sets the amount of time, in microseconds, that should elapse after starting an event before the single strobe should have a rising edge.

void ***setStrobeWidth*** (long deviceID, int% errorCode, unsigned long microseconds)

- Sets the amount of time, in microseconds, that the single strobe pulse should remain high after it begins.

bool ***getStrobeEnable*** (long deviceID, int% errorCode)

- Retrieves the given device's current strobe enable state.

unsigned long ***getStrobeDelay*** (long deviceID, int% errorCode)

- Retrieves the given device's current single strobe delay.

unsigned long ***getStrobeWidth*** (long deviceID, int% errorCode)

- Retrieves the given device's current single strobe pulse width.

unsigned long ***getStrobeMinimumDelay*** (long deviceID, int% errorCode)

- Retrieves the given device's minimum allowed single strobe delay.

unsigned long ***getStrobeMaximumDelay*** (long deviceID, int% errorCode)

- Retrieves the given device's maximum allowed single strobe delay.

unsigned long ***getStrobeMinimumWidth*** (long deviceID, int% errorCode)

- Retrieves the given device's minimum allowed single strobe pulse width.

unsigned long ***getStrobeMaximumWidth*** (long deviceID, int% errorCode)

- Retrieves the given device's maximum allowed single strobe pulse width.

unsigned long ***getStrobeMaximumCycle*** (long deviceID, int% errorCode)

- Retrieves the given device's maximum amount of time that the entire single strobe pulse can take. The sum of the pulse delay and width must never exceed this value. This is effectively the largest delay between the epoch and the falling edge that is allowed.

unsigned long ***getStrobeIncrementDelay*** (long deviceID, int% errorCode)

- Retrieves the given device's single strobe delay increment value.

unsigned long ***getStrobeIncrementWidth*** (long deviceID, int% errorCode)

- Retrieves the given device's single strobe pulse width increment value.

SpectrumProcess

unsigned char ***getBoxcarWidth*** (long deviceID, int% errorCode)

- Retrieves the given device's current boxcar filter width setting.

unsigned short int ***getScansToAverage*** (long deviceID, int% errorCode)

- Retrieves the given device's current onboard averaging setting.

void ***setBoxcarWidth*** (long deviceID, int% errorCode, unsigned char boxcarWidth)

- Sets the given device's onboard boxcar filter width.

void ***setScansToAverage*** (long deviceID, int% errorCode, unsigned short int scansToAverage)

- Sets the number of scans to be averaged per measurement for a given device.

void ***setTriggerMode*** (long deviceID, int% errorCode, int mode)

- Sets the device's trigger mode.

SpectrumWithMetadata

SpectrumWithMetadata (const oceandirect::SpectrumWithMetadata &nativeData)

- Wrapper class to contain raw spectrum with timestamp. This class may return 0 to 15 spectra with timestamps depending on what has accumulated in the data buffer of the spectrometer. In the .NET environment, the function returns a class with “properties” for the spectrum and timestamp.

Temperature

int ***getCount*** (long deviceID, int% errorCode)

- Retrieves the number of temperature sensors available on the given device.

double ***getTemperature*** (long deviceID, int% errorCode, int index)

- Retrieves the current temperature as reported by the specified temperature sensor on the given device.

ThermoElectric

double ***readTemperatureDegreesC*** (long deviceID, int% errorCode)

- Reads the current temperature of the thermoelectric cooler (TEC) on a given device (not the setpoint).

void ***setTemperatureSetpointDegreesC*** (long deviceID, int% errorCode, double temperatureDegreesCelsius)

- Sets the thermoelectric cooler (TEC) setpoint on the given device, which is the target temperature it will try to reach.

void ***setEnable*** (long deviceID, int% errorCode, unsigned char tecEnable)

- Enables/disables the thermoelectric cooler (TEC) on the given device.

bool ***getEnable*** (long deviceID, int% errorCode)

- Returns true if the thermoelectric cooler (TEC) on a given device has been enabled.

float ***getSetpoint*** (long deviceID, int% errorCode)

- Reads the setpoint temperature of the Thermoelectric Cooler (TEC) on a given device.

bool ***getStable*** (long deviceID, int% errorCode)

- Returns true if the thermoelectric cooler (TEC) on a given device has reached the the setpoint temperature and is not changing.

bool ***getFanEnable*** (long deviceID, int% errorCode)

- Returns true if the thermoelectric cooler (TEC) fan on a given device has been enabled.

void ***setFanEnable*** (long deviceID, int% errorCode, bool fanEnable)

- Enables/disables the thermoelectric cooler (TEC) fan on the given device.

Appendix B - Error Codes

The following are error codes that may be returned from a function call.

Error Code	Description	Error Code	Description
0	Successful/no error	16	Empty Vector (Check Input Parameter)
1	Undefined error	17	Color Conversion Error (Check Values)
2	No device found	18	No Peak Found Error (Input Spectrum Has No Peaks)
3	Could not close device	19	Illegal State Error (Unexpected State)
4	Feature not implemented	20	Minimum Integration Time Reached (Lamp is too bright)
5	No such feature on device	21	Maximum Integration Time Reached (Lamp is too dim)
6	Data transfer error	22	Please ensure that your lamp is on
7	Invalid user buffer provided	23	Not enough buffer space
8	Input was out of bounds	24	Command not supported by device
9	Spectrometer was saturated	25	Trial license expired
10	Value not found	26	No valid license
11	Divide By Zero Error (Cannot Divide by Zero)	27	License not checked
12	Non-Invertible Matrix Error (Matrix Has No Inverse)	28	Full license exceeded
13	Array Length Error (Array Lengths Don't Match)	29	Trial expired
14	Array Index Out of Bounds (Check Your Array Length? Is it Zero?)	30	Perpetual license version mismatch
15	Invalid Argument (Check Input Parameter)		

Appendix C - Improving Performance

Windows is not a real-time operating system and cannot guarantee a level of responsiveness. But there are a few things you can do to improve the performance of your application.

- You can increase the priority of the thread in which getSpectrum() is called. However, this only affects the priority of that thread WITHIN your application, and not relative to OTHER applications running on Windows. Often the problem is that some *other* application (or Windows service) is performing activity that interferes with the speed of your application. Try setting the priority of your OceanDirect application to "RealTime." To do this:

    1. Type control+alt+delete to bring up the Windows Task Manager.
    2. Select the **Processes** tab.
    3. Right-click on your OceanDirect application and choose **Set Priority | RealTime**.

- Determine what applications and services are running on your PC and shut down all unnecessary applications. If you have a backup utility such as Carbonite, you should pause it. Check your anti-virus application to see if it is configured in some way that might result in bursts of disk I/O.
- If bursts of disk I/O are interfering with your application, there is a good chance this disk I/O is due to "page faults". Page faults occur when Windows does not have enough RAM/memory to run all of the applications that are currently active. So Windows “borrows” some disk space to "simulate" additional RAM. If this causes your OceanDirect application a problem, then the solution is to either shut down as many applications as possible, or to install additional RAM.
- Try running your application on another PC that has nothing else installed.

Appendix D - FAQs

How fast can I acquire spectra?

The speed at which you can acquire spectra depends on the following factors:

- Minimum integration time.
- Communication speed of the USB connection. Most PCs use USB 3.0, but some may still be using USB 2.0, which is much slower.
- Number of pixels of data. Some detectors return as few as 256 pixels of data, others may return up to 4096 pixels. Each pixel is transmitted as a 2-byte integer. The greater the number of pixels, the longer it takes to transmit the data.
- Speed of your PC.
- The number of spectrometers operating in parallel.

Consider each of these factors carefully when estimating the maximum rate at which you can acquire spectra on your PC. It is always best to perform actual real-world measurements.

Why doesn't getSpectrum() return when I think it should?

When you set the integration time for a spectrometer, and then call *getSpectrum(),* you might expect that the duration of the call to *getSpectrum()* would match the integration time exactly. This is rarely the case, and there are several reasons why:

- When you first power up the spectrometer by plugging it in to your PC, the spectrometer immediately begins acquiring spectra using default settings for integration time and other acquisition parameters. This is called Normal mode. As long as the spectrometer remains in Normal mode, and remains plugged in to your PC, it will continuously acquire spectra.

  When your application calls *getSpectrum(),* the spectrometer may be just beginning to acquire a new spectrum, or it may be nearly finished acquiring a spectrum, or it may be anywhere in between these two extremes. As a consequence, *getSpectrum()* may return almost immediately, or it may not return until the full integration period has elapsed; it just depends on how far along the spectrometer happened to be in its acquisition process at the moment you called *getSpectrum()*.

  If your application calls *getSpectrum()* repeatedly (in a tight loop), the duration of subsequent calls to *getSpectrum()* will exactly match the integration time. This is because once you have called *getSpectrum(),* you are effectively synchronized with the operation of the spectrometer. Your first call to *getSpectrum()* will return at the exact moment that the

spectrometer has completed a spectrum acquisition. Your second call to *getSpectrum()* occurs just as the spectrometer is beginning the next spectrum acquisition. Thus, your second (and all subsequent) calls to *getSpectrum()* must wait for the full integration period to elapse before the spectrometer has finished acquiring the next spectrum.

- There is a second reason why the duration of the *getSpectrum()* method may not match the integration time setting. Whenever you change one of the acquisition parameters that can affect the spectrum data (e.g., changing the integration time), OceanDirect will automatically take a stability scan. This means OceanDirect will ignore the spectrum that was being acquired at the moment when you called *getSpectrum()*. This spectrum was acquired using previous settings for the acquisition parameters, and thus is invalid. OceanDirect will return the *following* spectrum, which is based on the new acquisition parameter settings.

If I change spectrometer models, do I need to change my program?

No. As long as you stay within the Wrapper API, your software does not need to be modified when you change to a different type of Ocean Insight spectrometer.

However, keep in mind that certain features (e.g., “thermoelectric cooling”) are only available on selected spectrometers.

What happens if the timeout period is shorter than the integration time?

Some of the calls to *getSpectrum()* will time out, and some will return valid spectra. The ratio of valid spectra to timeouts will depend on the ratio of your timeout period to the integration time. However, this situation should not cause you to miss spectral acquisitions, unless the integration time is so short your program cannot call *getSpectrum()* soon enough to capture the next spectrum.

What happens in the following scenario?

1. Set spectrometer to one of the external trigger modes
2. Call *getSpectrum()*
3. Timeout occurs before the trigger occurs
4. Trigger event happens before you call getSpectrum() a second time
5. Call *getSpectrum()*

The second call to *getSpectrum()* will return immediately with the spectrum that was acquired when the trigger event occurred.

If multiple trigger events happen after the first *getSpectrum()* has timed-out and before you call getSpectrum() a second time, when you do call getSpectrum() the second time, it immediately returns with the spectrum from the FIRST trigger event. Data from any additional trigger events during this window will be lost.

If a timeout occurs before the spectrum can be acquired, the spectral array returned by *getSpectrum()* will contain all zeroes.

How do I use Ethernet to send commands to my spectrometer?

The OceanDirect commands can be executed through an Ethernet interface on Ocean FX and Ocean HDX spectrometers. First, however, the spectrometer must be configured using the USB connection to obtain the IP address. Please refer to the Ocean FX or Ocean HDX User Manuals Setup and Installation section to obtain the IP address.

Unlock the Unknown

Ocean Insight exists to end guessing. We equip humanity with technology and data to make precisely informed decisions providing transformational clarity for human advancement in health, safety, and the environment.

Questions?
Chat with us at **OceanInsight.com**.
info@oceaninsight.com • **US** +1 727-733-2447
EUROPE +31 26-3190500 • **ASIA** +86 21-6295-6600