StellarNet Universal Driver SDK Developer Guide

Version 1.0

14390 Carlson Circle
Tampa, FL 33626
+1(813) 855-8687

Table of Contents

Introduction

The StellarNet SDK provides a powerful, flexible, and platform independent interface for integrating StellarNet devices into your applications. Delivered as compiled dynamic libraries (DLLs on Windows, .so on Linux, and .dylib on macOS), it supports both 32 bit and 64 bit architectures and works with a wide range of programming environments that can call dynamic libraries, including C, Python, C#/.NET, Java, Excel VBA, MATLAB, and more, giving developers the freedom to use the tools and platforms they prefer. Designed to accelerate development and unlock the full potential of your StellarNet hardware, the SDK offers ready to use functions for device control, data acquisition, and spectral analysis, while included example projects demonstrate common usage patterns and serve as a starting point for building custom solutions tailored to your specific applications and workflows.

Key capabilities include

- Discovering and connecting to spectrometer devices via USB
- Configuring acquisition settings such as integration time, averaging, smoothing, timing mode, and temperature compensation
- Acquiring raw spectral data, including wavelength and intensity arrays
- Performing high speed burst acquisition modes when supported by the hardware
- Applying radiometric and analytical transformations such as Scope, Transmission, Absorbance, Watts, Lumens, and PAR

Compiled API Support

This section lists the compiled drivers provided for each platform and architecture. It includes their dependencies and notes on usage to ensure proper integration with your applications.

Driver File	OS	Architecture	Dependency	Description
Windows/Windows_x86/StellarNetAPI_windows_x86.dll	Windows	x86 (32-bit)	libusb-1.0.dll (32-bit)	32-bit Windows driver for StellarNet devices. Requires the 32-bit libusb runtime DLL to be located in the same folder or accessible via the system PATH.
Windows/Windows_x64/StellarNetAPI_windows_x64.dll	Windows	x64 (64-bit)	libusb-1.0.dll (64-bit)	64-bit Windows driver for modern applications. Requires the 64-bit libusb runtime DLL to be located in the same folder or accessible via the system PATH.
Linux/StellarNetAPI_linux_x64.so	Linux	x86_64 (64-bit)	libusb-1.0 (system package)	64-bit Linux shared library. Requires libusb to be installed via the system package manager (e.g., apt, yum).
Linux/StellarNetAPI_linux_i386.so	Linux	i386 (32-bits)	libusb-1.0 (system package)	32-bit Linux shared library. Requires libusb to be installed via the system package manager (e.g., apt, yum).
macOS/StellarNetAPI_macOS_arm64.dylib	macOS	ARM64 (Apple Silicon)	libusb (Homebrew)	Apple Silicon macOS library. Requires libusb installed, commonly via Homebrew.
macOS/StellarNetAPI_macOS_x86_64.dylib	macOS	x86_64 (Intel)	libusb (Homebrew)	Intel macOS library. Requires libusb installed, commonly via Homebrew.

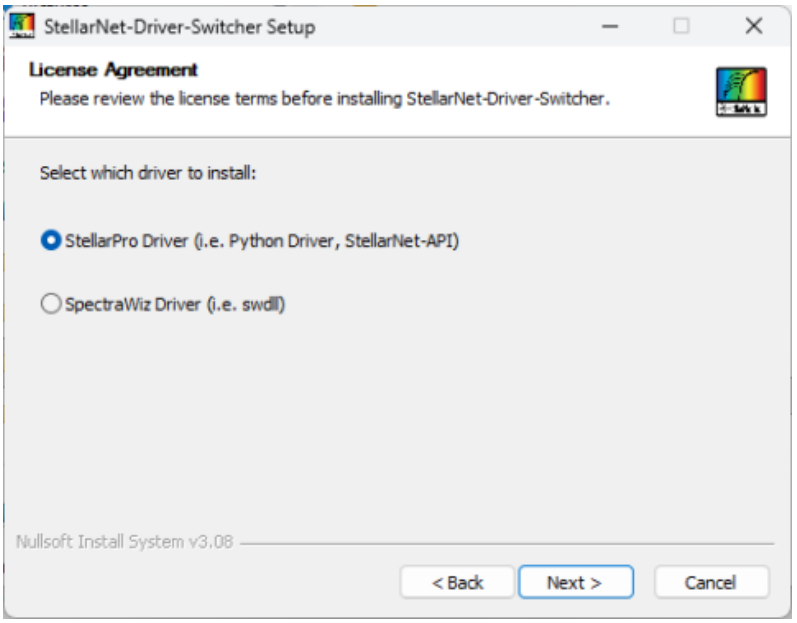
StellarNet SDK Driver Installation

Windows Installation

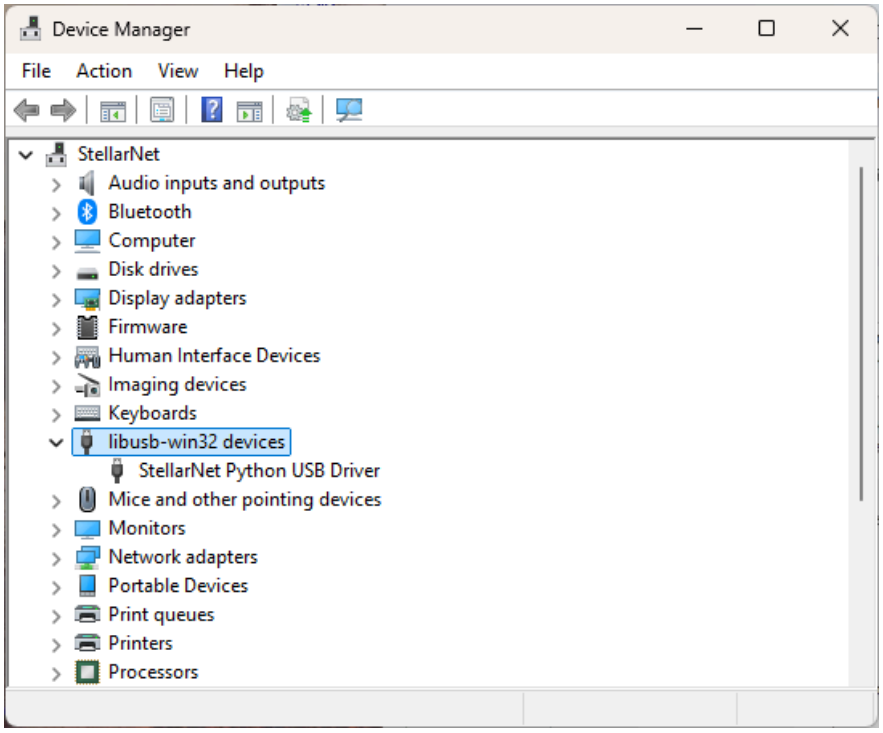
This StellarNet SDK operates using the **StellarPro** driver called "**StellarNet Python USB Driver**". It is important to note that Windows cannot actively run both the **SpectraWiz (USBDEV > StellarNet spectrometer)** and **StellarPro (StellarNet Python USB Driver)** drivers simultaneously. You must ensure the correct driver is active for your specific software environment.

Installation Steps

1. **Connect Hardware:** Ensure the spectrometer is connected to the computer via USB.
2. **Launch Switcher:** Locate and double-click ***StellarNet-Driver-Switcher.exe*** in the Windows directory, granting Administrative (UAC) permissions when prompted.
3. **Select Driver:** Within the utility interface, select the **StellarPro Driver (i.e. Python Driver, StellarNet-API)** and click **Next**.



4. **Execute and Verify:** Click **Install** and follow the prompts to complete the driver switching process. After the installation is complete, you should see **libusb-win32 devices → StellarNet Python USB Driver** in **Device Manager** when the spectrometer is connected. You will then be able to use the compiled driver file.



Note: If you need to return to SpectraWiz-based applications later, simply re-run the switcher utility and select the **SpectraWiz** driver option. For further technical details and manual driver switching steps, refer to the StellarNet Knowledge Base.

---

macOS Installation

Unlike Windows, macOS environments do not require a driver switching utility. Instead, the system utilizes the **libusb** library to communicate directly with StellarNet spectrometer hardware.

1. **Open Terminal:** Locate the Terminal application via Spotlight search or within your Applications/Utilities folder.
2. **Install libusb:** On macOS, the Homebrew package manager is the standard method for installation. Execute the command: ***brew install libusb***

Linux Installation

Unlike Windows, Linux environments do not require a driver switching utility. Instead, these systems utilize the **libusb** library to communicate directly with StellarNet spectrometer hardware.

1. **Open Terminal:** Access your command-line interface.

2. **Update Package Lists:** Use the **APT** package handling utility to refresh your local package index by executing: ***sudo apt-get update***

3. **Install libusb:** Install the required development headers for the library via **APT** by executing: ***sudo apt-get install libusb-1.0-0-dev***

4. **Permissions (Optional):** Depending on your distribution, you may need to add a ***udev*** rule to allow the SDK to access the USB device without root privileges.

Directory Structure

This section provides an overview of the files and folders included in the StellarNet SDK. It describes the location of header files, compiled drivers, and example projects for each supported language.

- **StellarNetAPI.h**

  Main API header file that provides the public interface for accessing the compiled dynamic libraries. It defines all the functions and data structures necessary for controlling StellarNet spectrometers and acquiring spectral data.

- **Windows**

  The StellarNet SDK provides architecture-specific libraries that require the libusb runtime to be present in the application's working directory or the system PATH; please note that while the provided demo examples default to these locations, you must manually update the library path variables within the source code if those library files are relocated.

  - **Windows_x86**
    - **StellarNetAPI_windows_x86.dll** – 32-bit Windows driver for StellarNet devices. Requires 32-bit libusb-1.0.dll in the same folder or system PATH.
    - **libusb-1.0.dll** – 32-bit libusb runtime DLL.
  - **Windows_x64**
    - **StellarNetAPI_windows_x64.dll** – 64-bit Windows driver for modern applications. Requires 64-bit libusb-1.0.dll in the same folder or system PATH.
    - **libusb-1.0.dll** – 64-bit libusb runtime DLL.
  - **StellarNet-Driver-Switcher.exe** – An automated utility that allows users to manually select between StellarNet drivers. Once a selection is made, the tool automatically performs the necessary driver removal and installation steps. **Ensure the StellarPro Driver (i.e., Python Driver, StellarNet-API) is selected when using this SDK.**

- **Linux**
  - **StellarNetAPI_linux_x64.so** – 64-bit Linux shared library. Requires libusb installed via system package manager (apt, yum, etc.).

- **macOS**
  - **StellarNetAPI_macOS_arm64.dylib** – Apple Silicon macOS library. Requires libusb installed (commonly via Homebrew).
  - **StellarNetAPI_macOS_x86_64.dylib** – Intel macOS library. Requires libusb installed (commonly via Homebrew).

- **example_c**
    - **StellarNet_Demo.c** – Demonstrates calling the compiled dynamic library in pure C.
    - **StellarNetAPILoader.c** – Helper file for loading and calling library functions.
    - **StellarNetAPILoader.h** – Header defining functions for accessing the compiled library.
- **example_python**
    - **StellarNet_Demo.py** – Demonstrates using the compiled dynamic library via Python.
    - **StellarNetAPILoader.py** – Python file for loading library functions and providing access.
- **example_csharp-dotnet**
    - **StellarNet_Demo.cs** – Demonstrates calling the compiled dynamic library in C#/.NET.
    - **StellarNetAPILoader.cs** – C# file for loading library functions and providing access.
- **example_java**
    - **StellarNet_Demo.java** – Demonstrates using the compiled dynamic library via Java.
    - **StellarNetAPILoader.java** – Java file for loading library functions and providing access.
- **example_excelVBA**
    - **StellarNet_Demo.xlsm** – Excel file containing VBA macros that call the compiled dynamic library.
        - **StellarNet_ExcelVBA_Demo** – Demonstrates using the compiled dynamic library in ExcelVBA.
        - **StellarNetAPILoader** – Module loading library functions and providing access.
- **Example_labview**
    - **StellarNet_Demo.vi** – Demonstrates calling the compiled dynamic library in LabVIEW using the Call Library Function Node.
- **DemoTest.CAL**
    - Demo StellarNet calibration file for radiometry data, provided as an example file for development and testing purposes.

Getting Started With the Demo Examples

The included demo projects demonstrate how to use the StellarNet SDK in different programming environments. Each example contains independent functions that illustrate specific command groups, such as device initialization, spectral data acquisition, instrument configuration, multi-device handling, and advanced spectral processing. These examples are intended for testing, reference, and as a foundation for custom development.

Key points to keep in mind when running the provided demos:

- **Device Lifecycle Management:** For simplicity and isolation, each example function opens the spectrometer at the start and closes it upon completion. In production environments, it is recommended to initialize the device once at startup and maintain the connection until the application is terminated.
- **Directory Integrity:** To ensure the examples run without additional configuration, do not move, rename, or rearrange the files within the provided folder structure.
- **Automated Library Loading:** The SDK is designed to automatically load the appropriate StellarNet shared library based on your operating system and architecture, provided the directory structure remains intact. The demo examples are configured to reference library files within their respective Windows, macOS, or Linux folders; if you relocate these files, you must manually update the path variables within the source code.
- **Hardware Troubleshooting:** If a device is not detected, verify that the driver is in the correct directory, the USB connection is stable, and the appropriate system permissions are granted (particularly for Linux *udev* rules or macOS security settings).

The examples serve as a starting point, and developers are encouraged to modify them or extract specific functions as building blocks for their own applications in Python, C, C#, Java, Excel VBA, or other supported programming environments.

Language	Directory	Execution Instructions
Python	example_python	**Windows:** python StellarNet_Demo.py **macOS:** python3 StellarNet_Demo.py **Linux:** sudo python3 StellarNet_Demo.py
C	example_c	**Windows:** gcc -std=c99 StellarNet_Demo.c StellarNetAPILoader.c -o StellarNet_Demo&& StellarNet_Demo.exe **macOS:** gcc -std=c99 StellarNet_Demo.c StellarNetAPILoader.c -o StellarNet_Demo-ldl && ./StellarNet_Demo **Linux:** gcc -std=c99 StellarNet_Demo.c StellarNetAPILoader.c -o StellarNet_Demo -lpthread -ldl && sudo ./StellarNet_Demo
C# / .NET	example_csharp-dotnet	**Windows:** dotnet run **macOS:** dotnet run
Java	example_java	**Windows:** javac --add-modules ALL-SYSTEM -cp ".;jna-jpms-5.18.1.jar" StellarNetAPILoader.java StellarNet_Demo.java && java -cp ".;jna-jpms-5.18.1.jar" StellarNet_Demo **macOS:** javac --add-modules ALL-SYSTEM -cp ".:jna-jpms-5.18.1.jar" StellarNetAPILoader.java StellarNet_Demo.java && java --enable-native-access=ALL-UNNAMED -cp ".:jna-jpms-5.18.1.jar" StellarNet_Demo
Excel VBA	example_excelVBA	**Windows:** Open the **StellarNet_Demo** spreadsheet and execute the macros or click the integrated buttons to perform specific tasks. (requires enabling Developer Mode)
LabVIEW	example_labview	Use the **LabVIEW Call Library Function Node** to load the **StellarNetAPI** library, ensuring the selected library bitness matches your LabVIEW environment, and then run the provided example VI.
MATLAB	example_matlab	Open and execute example.m directly within the MATLAB editor.


StellarNet Inc

API Reference

This section contains detailed information on all functions available through the StellarNet SDK. For each function, you will find its signature, parameters, and return values to help integrate it into your own applications.

Error Codes

Code	Name	Description
0	DEV_SUCCESS	Operation succeeded
-1	DEV_ERROR_NULL	Invalid or null device handle
-2	DEV_ERROR_INVALID_ARG	Invalid argument or parameter
-3	DEV_ERROR_TIMEOUT	USB communication timeout
-4	DEV_ERROR_USB	USB communication failure
-5	DEV_ERROR_ALLOC	Memory allocation failure
-6	DEV_ERROR_EEPROM	EEPROM read/write failure
-7	DEV_ERROR_RANGE	Parameter out of valid range
-8	DEV_ERROR_UNSUPPORTED	Feature not supported
-9	DEV_ERROR_UNKNOWN	Unknown internal error
-10	DEV_ERROR_CALFILE	Calibration file error
-11	DEV_ERROR_GAIN_TABLE	Gain table processing failure
-12	DEV_ERROR_NOT_FOUND	Device not found
-13	DEV_ERROR_CSF_NOT_FOUND	Calibration source file not found
-14	DEV_ERROR_BUFFER_TOO_SMALL	Output buffer too small

Device Handle

typedef void* StellarNetDeviceHandle;

A StellarNetDeviceHandle is an opaque pointer representing a connected spectrometer device. It encapsulates the internal state of the device and must only be accessed via API functions. Direct manipulation of the pointer is not allowed.

Functions that return a device handle:

Function	Description
void* get_device(uint8_t channel);	Opens and initializes a spectrometer device by index. Returns a handle to the device or NULL if the device could not be opened.
void* get_device_with_wavelength(uint8_t channel, double* wavelength_buffer);	Opens a device and immediately loads its wavelength calibration into the provided buffer. Returns a handle or NULL on failure.
int get_all_devices(int* device_count, uintptr_t* devs);	Retrieves handles for all currently connected devices. Populates the devs array with uintptr_t values that can be cast to StellarNetDeviceHandle. Returns a DevError status code.

Important Notes:

- A device handle returned by these functions must be reused for all operations on that device until it is closed.
- To release the handle and free resources, call: int close_device(void *dev);

- Never use a handle after close_device() has been called. Doing so results in undefined behavior.

Device Management & Discovery

get_device

void* get_device(uint8_t channel);

Description:
Opens and initializes a spectrometer device for the specified channel. This establishes communication with the hardware and prepares it for configuration and spectral acquisition. This function should be called once per device at application startup. The returned handle must be reused for all subsequent operations until the device is closed.

Parameters:

- channel (uint8_t) – Zero-based index of the spectrometer
  - 0 = first device
  - 1 = second device
  - (etc.)

Returns:

- void* – A valid device handle
- NULL – If the device could not be opened or initialized.

Usage Notes:

- Call the function once per device during the application startup phase.
- Store and reuse the returned handle for all subsequent API operations.
- Avoid repeatedly opening the same device to prevent communication conflicts.
- Always release the handle using close_device() to properly free system resources.

---

get_device_with_wavelength

void* get_device_with_wavelength(uint8_t channel, double* wavelength_buffer);

Description:
Opens and initializes a spectrometer device and immediately loads the wavelength into the provided buffer. This combined operation simplifies the startup sequence when the wavelength axis is required for immediate data processing.

Parameters:

- channel (uint8_t) – Zero-based index of the spectrometer.
- wavelength_buffer (double*) – A preallocated array where the wavelength will be stored.

Returns:

- void* – A valid device handle.

- NULL – If the device could not be opened or initialized.

Usage Notes:

- Similar to **get_device**, call this function once per device during the application startup phase and reuse the returned handle for all subsequent API operations.
- Ensure the buffer is preallocated based on the specific pixel count of the device.
- This is the preferred initialization method when the wavelength axis is needed immediately after connection.

---

get_all_devices

int get_all_devices(int* device_count, uintptr_t* devs);

Description:
Retrieves handles for all currently connected spectrometer devices in a single operation. This function is designed for managing multiple instruments dynamically within a single application.

Parameters:

- device_count (int*) – Output pointer that stores the total number of spectrometers detected.
- devs (uintptr_t*) – A preallocated array that will be populated with the individual device handles.

Returns:

- DevError status code (int)

Usage Notes:

- Ensure the devs array is large enough to accommodate all potential devices.
- This function is ideal for complex systems requiring synchronized multi-device management.

---

total_device_count

int total_device_count();

Description:
Returns the total number of spectrometer devices currently connected to the system.

Returns:

- int – The count of detected devices.

Usage Notes:

- This is a lightweight check that does not perform handle allocation or device initialization.
- Use this function to determine the required size for the devs array before calling **get_all_devices()**, providing a reliable way to verify hardware presence before starting the application setup sequence.

---

is_device_connected

int is_device_connected(void* dev);

Description:
Checks whether a specific device handle is still valid and responsive to communication. This is used to verify that the hardware is still physically connected and powered.

Parameters:

- dev (void*) – The device handle to be verified.

Returns:

- DevError status code (int)

Usage Notes:

- Use this function before starting an acquisition in long-running applications to ensure the hardware is still available.
- This check helps detect unexpected USB disconnects or power loss during operation.

---

close_device

int close_device(void* dev);

Description:
Closes the spectrometer device and releases all allocated system resources. After this function is executed, the provided handle becomes invalid and cannot be used for further operations.

Parameters:

- dev (void*) – Device handle

Returns:

- DevError status code (int)

Usage Notes:

- Call this function once per device during the application shutdown sequence.
- Never attempt to reuse a handle after closing it, as this will result in communication errors.
- This call is required to properly free USB resources and make the device available for other applications.

---

shutdown_usb

int shutdown_usb();

Description:
Shuts down the entire USB subsystem and releases all remaining device-level resources. This serves as a global cleanup command for the SDK.

Returns:

- DevError status code (int)

Usage Notes:

- Call this function once at the very end of the application exit process.
- Using this function prevents USB lockups and potential driver issues upon subsequent application launches.
- This should be the final SDK call made by your software (after **close_device** for each device) before termination.

Device Information

getPixelCount

uint16_t getPixelCount(void* dev);

Description:
Returns the number of detector pixels for the specific spectrometer. This value determines the required size for all spectral data buffers, including wavelength and intensity arrays.

Parameters:

- dev (void*) – The active device handle.

Returns:

- uint16_t – The total pixel count of the detector.

Usage Notes:

- Call this function before allocating any buffers for spectral data to ensure they are the correct size.
- Use the returned value to dynamically manage memory allocation for various data processing tasks.

---

getSerialNumber

const char* getSerialNumber(void* dev);

Description:
Retrieves the unique hardware serial number assigned to the spectrometer.

Parameters:

- dev (void*) – The active device handle.

Returns:

- const char* – A pointer to a null-terminated string containing the serial number.

Usage Notes:

- This function is particularly useful for logging, automated reporting, and tracking specific devices in multi-spectrometer systems.

---

getFullDeviceID

const char* getFullDeviceID(void *dev);

Description:

Retrieves a comprehensive identification string for the spectrometer. This typically includes the model name, detector type, and other factory-encoded descriptors.

Parameters:

- dev (void)* – The active device handle.

Returns:

- const char* – A pointer to a null-terminated string containing the full device identification details.

Usage Notes:

- This string provides more detailed hardware information than the serial number alone.
- It is commonly used for generating professional calibration reports and system diagnostics.

---

getCoeffs

const double* getCoeffs(void* dev);

Description:
Returns a pointer to the internal wavelength calibration coefficients stored in a 4-element array.

Parameters:

- dev (void*) – The active device handle.

Returns:

- const double* – A pointer to the four calibration coefficients.

Usage Notes:

- This function is commonly used for local wavelength reconstruction and data persistent storage.

---

getCoeffs_array

int getCoeffs_array(void* dev, double* coeffs_buffer);

Description:
Copies the wavelength calibration coefficients from the device into a user-provided buffer.

Parameters:

- dev (void*) – The active device handle.
- coeffs_buffer (double*) – A preallocated output array of at least size 4.

Returns:

- DevError status code (int)

Usage Notes:

- Ensure the provided buffer is at least size 4 to prevent memory overflow.
- This function is commonly used for local wavelength reconstruction and data persistent storage.

---

getIntegrationTime

uint32_t getIntegrationTime(void* dev);

Description:
Returns the current integration time (exposure duration) in milliseconds currently set for spectral acquisition for a specific device.

Parameters:

- dev (void*) – The active device handle.

Returns:

- uint32_t – Integration time in milliseconds.

Usage Notes:

- This value directly reflects the active hardware state and is useful for updating UI displays or providing metadata for data logging.

---

getXTiming

uint8_t getXTiming(void* dev);

Description:
Returns the current XTiming mode for a specific device.

Parameters:

- dev (void*) – The active device handle.

Returns:

- uint8_t – The timing mode (1–3).

Usage Notes:

- This value directly reflects the active hardware state and is useful for updating UI displays or providing metadata for data logging.

---

getSmooth

uint8_t getSmooth(void* dev);

Description:
Returns the current smoothing level applied to the spectral data for a specific device.

Parameters:

- dev (void*) – The active device handle.

Returns:

- uint8_t – The smoothing level (0–4)

Usage Notes:

- This value directly reflects the active hardware state and is useful for updating UI displays or providing metadata for data logging.

---

getScansToAvg

uint32_t getScansToAvg(void* dev);

Description:
Returns the number of individual scans that are averaged together to produce a single measurement.

Parameters:

- dev (void*) – The active device handle.

Returns:

- uint32_t – The current scan averaging count.

Usage Notes:

- This value directly reflects the active hardware state and is useful for updating UI displays or providing metadata for data logging.

---

getTrigger

uint8_t getTrigger(void* dev);

Description:
Returns the current hardware trigger mode setting for a specific device.

Parameters:

- dev (void*) – The active device handle.

Returns:

- uint8_t – The active trigger mode.

Usage Notes:

- This value directly reflects the active hardware state and is useful for updating UI displays or providing metadata for data logging.

---

is_zapFeatureAvailable

int is_zapFeatureAvailable(void *dev);

Description:

Checks whether the connected spectrometer supports the ZAP electronics and burst mode features.

Parameters:

- dev (void)* – The active device handle.

Returns:

- int – Returns `1` if the ZAP feature is supported, or `0` if it is not supported.

Usage Notes:

- Call this function before attempting to use high-speed burst functions like **getBurstFifo_Y**.
- This allows the application to dynamically adjust its UI or capabilities based on the specific hardware model connected.

Spectral Acquisition

getSpectrum_X

int getSpectrum_X(void* dev, double* output_buffer);

Description:
Retrieves the wavelength values in nanometers corresponding to each detector pixel. These values form the X-axis (wavelength scale) for your spectral data.

Parameters:

- dev (void*) – The active device handle.
- output_buffer (double*) – A preallocated array where the wavelength values will be stored.

Returns:

- DevError status code (int)

Usage Notes:

- The output buffer must be preallocated to a size that matches the device pixel count.

---

getSpectrum_Y

int getSpectrum_Y(void* dev, double* output_buffer);

Description:
Retrieves the intensity values for each pixel from the most recent scan. This represents the magnitude of light detected at each wavelength.

Parameters:

- dev (void*) – The active device handle.
- output_buffer (double*) – A preallocated array to store the intensity data.

Returns:

- DevError status code (int)

Usage Notes:

- Intensity data must be paired with wavelength data (from **getSpectrum_X** or **get_device_with_wavelength**) to reconstruct a complete spectrum.
- The resulting values are directly affected by the active hardware settings, including integration time, gain, and smoothing levels.

---

getBurstFifo_Y

int getBurstFifo_Y(void* dev, uint16_t* output_buffer);

Description:
Retrieves high-speed burst acquisition data stored in the device's FIFO memory. This is used for capturing rapid transient events that exceed standard communication speeds.

Parameters:

- dev (void*) – Device handle
- output_buffer (uint16_t*) – Preallocated burst buffer. The provided buffer must be large enough to hold the entire burst sequence, typically calculated as the burst size (e.g., 128 scans) multiplied by the pixel count.

Returns:

- DevError status code (int)

Usage Notes:

- Only supported on compatible devices. Device compatibility can be checked with **is_zapFeatureAvailable**.

Device Configuration

setParams

int setParams(void* dev, uint32_t inttime, uint32_t scansavg, uint8_t smooth, uint8_t xtiming, uint8_t temp_comp, uint8_t clear);

Description:
Sets multiple acquisition parameters in a single call. This is the preferred method for configuring the device efficiently.

Parameters:

- dev (void*) – The active device handle.
- inttime (uint32_t) – Integration time in milliseconds.
- scansavg (uint32_t) – Number of scans to average (1…).
- smooth (uint8_t) – Digital smoothing level (0–4).
- xtiming (uint8_t) – XTiming mode (1–3).
- temp_comp (uint8_t) – Temperature compensation (0 = Off, 1 = On).
- clear (uint8_t) – Reset flag to clear the first scan from the internal buffer. (0 = No, 1 = Yes). Using the clear flag is recommended when changing parameters to ensure the next acquired spectrum strictly reflects the new settings. **Note:** With the clear flag set to 1 (True), the hardware will require a duration equal to the integration time × scans to average to complete the first data acquisition and populate the buffer with fresh data.

Parameters Info

- **Detector integration time**:
This should be adjusted for each experiment to maximize the detector output and signal to noise ratio.

- **Number of scans to average**: (1…)
Sets the number of spectra to signal average. Please note that the real-time display is updated AFTER these numbers of spectra have been acquired. This option provides a smoothing in the Y-axis, effectively increasing the system signal to noise by the square root of the number of scans being averaged. Set the averaging to the highest number

- **Smoothing level** 0, 1, 2, 3, or 4
This performs data smoothing by applying a moving average of adjacent pixels to the data arrays. For example, a Pixel Boxcar setting of 1 would average 5 total pixels: 2 pixels on the left, 2 pixels on the right, and one in the center. 0 performs NO data smoothing.

Smoothing Level	Total pixels averaged
0	0

1	5
2	9
3	17
4	33

- **Temperature compensation**:

The first 15 "optical black" pixels that are read from the detector, except Hamamatsu and InGas, provide a useful level of the sensor dark current even when the detector is illuminated. If this option is on, the system periodically samples this area and makes an adjustment to the normal readout. When the level raises or lowers with temperature, the scan is adjusted accordingly.

- **XTiming resolution control**: 1/2/3

This feature provides increased optical resolutions. Selection 1 is the lowest optical resolution and is synchronized with the selected detector integration. In general, if your requirements for optical resolution are greater than 1nm, then selection 1 is ok. Selection 2 or 3 slows the digitizer & detector clock by a factor of 2 and 4 respectively. With XT levels 2 & 3 you will be able to observe increasingly higher resolutions. The detectors signal amplifier improves with slower throughput. When selecting XT level 2/3 the detector integration time must be increased to 30ms or longer to avoid sync delays.

Returns:

- DevError status code (int)

Usage Notes:

- This function is preferred over calling multiple individual setter functions to improve application performance and ensure parameter synchronization.

---

setIntegrationTime

int setIntegrationTime(void* dev, uint32_t int_time);

Description:
Sets the detector integration time (exposure duration). This controls how long the spectrometer collects light before reading out a spectrum. It is one of the most important parameters affecting signal intensity and saturation.

Parameters:

- dev (void*) – The active device handle.
- int_time (uint32_t) – Integration time in milliseconds. Higher values increase signal intensity but may lead to detector saturation; lower values reduce signal but allow for a higher dynamic range in bright environments. This should be adjusted for each experiment to maximize the detector output and signal to noise ratio.

Returns:

- DevError status code (int)

---

setXTiming

int setXTiming(void* dev, uint8_t x_timing);

Description:
Sets the XTiming resolution control, adjusting the detector timing resolution used during spectral acquisition and measurement operation.

Parameters:

- dev (void*) – The active device handle.
- x_timing (uint8_t) – XTiming resolution control (1–3).
    - Selection 1 provides the lowest optical resolution and syncs with the selected detector integration time. For applications requiring resolution above 1 nm, Selection 1 is typically sufficient. Selections 2 and 3 slow the digitizer and detector clock by factors of 2 and 4, improving optical resolution and signal amplification. When using XT levels 2 or 3, the detector integration time must be set to 30 ms or longer to avoid synchronization delays.

Returns:

- DevError status code (int)

---

setSmooth

int setSmooth(void* dev, uint8_t x_smooth);

Description:
Sets digital smoothing level applied to spectral data to reduce noise.

Parameters:

- dev (void*) – The active device handle.
- x_smooth (uint8_t) – Smoothing level (0–4). A value of 0 provides raw data with no smoothing. While higher values effectively reduce noise in low-light conditions, they also slightly decrease the effective spectral resolution.
    - This performs data smoothing by applying a moving average of adjacent pixels to the data arrays. For example, a Pixel Boxcar setting of 1 would average 5 total pixels: 2 pixels on the left, 2 pixels on the right, and one in the center. 0 performs NO data smoothing.

Smoothing Level	Total pixels averaged
0	0
1	5
2	9
3	17
4	33

Returns:

- DevError status code (int)

---

setScansToAvg

int setScansToAvg(void* dev, uint32_t scans_to_avg);

Description:
Sets the number of individual scans to be averaged together per measurement to improve the Signal-to-Noise Ratio (SNR).

Parameters:

- dev (void*) – The active device handle.
- scans_to_avg (uint32_t) – The number of scans to average. Higher averaging values improve data stability but reduce the overall measurement speed.
    - Sets the number of spectra to signal average. Please note that the real-time display is updated AFTER these numbers of spectra have been acquired. This option provides a smoothing in the Y-axis, effectively increasing the system signal to noise by the square root of the number of scans being averaged. This setting should be balanced with the integration time to achieve the desired signal quality within an acceptable timeframe.

Returns:

- DevError status code (int)

---

setTemperatureCompensation

int setTemperatureCompensation(void* dev, uint8_t temp_comp);

Description:
Enables or disables internal temperature compensation to mitigate spectral drift caused by fluctuations in detector temperature.

Parameters:

- dev (void*) – The active device handle.
- temp_comp (uint8_t) – 0 to disable, 1 to enable.
    - The first 15 "optical black" pixels that are read from the detector, except Hamamatsu and InGas, provide a useful level of the sensor dark current even when the detector is illuminated. If this option is on, the system periodically samples this area and makes an adjustment to the normal readout. When the level raises or lowers with temperature, the scan is adjusted accordingly.

Returns:

- DevError status code (int)

---

setTrigger

int setTrigger(void* dev, uint8_t trigger);

Description:
Enable or disable the external trigger for the selected spectrometer device. The timeout function will be disabled if the variable trigger is set to True. When the trigger is True, the timeout will be ignored, which may result in infinite waiting if the spectrometer gets disconnected or if data does not return for some reason. When the trigger is False (the default), a default timer will be set based on the integration time and the number of scans to average. If the data does not return after the timer completes, an error will be thrown to inform the user.

Parameters:

- dev (void*) – The active device handle.
- trigger (uint8_t) – The trigger mode selection (1=True, no timeout. 0 = False, with timeout).

Returns:

- DevError status code (int)

Usage Notes:

- Ensure the software trigger mode matches the physical electrical configuration of the external trigger source.

---

setTemporary_gain

int setTemporary_gain(void* dev, int gain_value);

Description:
Temporarily adjusts the detector's analog gain for short-term signal amplification. This is used to boost signal levels without modifying the factory-calibrated gain settings stored in the device's non-volatile memory (zAP2 devices only).

Parameters:

- dev (void*) – The active device handle.
- gain_value (int) – The desired gain level to apply to the detector (1-5).

Returns:

- DevError status code (int)

Usage Notes:

- **Support:** This function is only supported on hardware equipped with **ZAP2** electronics. Compatibility should be verified using **is_zapFeatureAvailable** before use.
- This setting is volatile; it does not permanently store the value and will reset upon power-cycling the device.

---

setTemporary_baseline

int setTemporary_baseline(void* dev, int baseline_count);

Description:
Temporarily sets the baseline offset to correct the detector's dark level prior to acquisition. This allows for real-time adjustment of the "floor" of the spectral data. (ZAP2 devices only).

Parameters:

- dev (void*) – The active device handle.
- baseline_count (int) – The baseline offset value (counts).

Returns:

- DevError status code (int)

Usage Notes:

- **Support:** This function is only supported on hardware equipped with **ZAP2** electronics. Compatibility should be verified using **is_zapFeatureAvailable** before use.
- **Dark Requirement:** When calling this function, ensure the spectrometer is completely shielded from light (looking at "dark"). No light should enter the spectrometer during this adjustment to ensure an accurate baseline.
- This setting is volatile; it does not permanently store the value and will reset upon power-cycling the device.

Spectral Calculations

getScopeY

```
int getScopeY(const double* spectrometerWavelength,
              const double* rawDarkData,
              const double* rawSampleData,
              size_t inputDataSize,
              double* output_buffer);
```

Description:
Computes a dark-subtracted raw spectrum (Scope mode). This function removes the dark signal from the sample signal, producing a corrected intensity spectrum that represents the true optical response.

Parameters:

- spectrometerWavelength (const double*) – Wavelength array corresponding to each pixel.
- rawDarkData (const double*) – The measured dark spectrum (taken with no light).
- rawSampleData (const double*) – The raw sample spectrum measurement.
- inputDataSize (size_t) – Total number of pixels in the input spectrum (i.e., the length of spectrometerWavelength)
- output_buffer (double*) – Preallocated array for the corrected scope spectrum. **The buffer size must be at least as large as the inputDataSize parameter.**

Returns:

- DevError status code (int)

---

getTransmissionY

```
int getTransmissionY(const double* spectrometerWavelength,
                     const double* rawDarkDataY,
                     const double* rawRefeDataY,
                     const double* rawSampleDataY,
                     size_t inputDataSize,
                     double* output_buffer);
```

Description:
Computes the transmission spectrum by comparing sample intensity against reference and dark measurements. The output represents the fraction of light transmitted through the sample.

Parameters:

- spectrometerWavelength (const double*) – Wavelength array corresponding to each pixel.
- rawDarkDataY (const double*) – Dark spectrum measurement.
- rawRefeDataY (const double*) – Light reference (100% baseline) spectrum.
- rawSampleDataY (const double*) – Sample spectrum measurement.
- inputDataSize (size_t) – Total number of pixels in the input spectrum (i.e., the length of spectrometerWavelength)
- output_buffer (double*) – A preallocated array for the resulting transmission values. **The buffer size must be at least as large as the inputDataSize parameter.** Output ranges typically from 0 to 100%.

Returns:

- DevError status code (int)

---

getAbsorbanceY

```
int getAbsorbanceY(const double* spectrometerWavelength,
                   const double* rawDarkDataY,
                   const double* rawRefeDataY,
                   const double* rawSampleDataY,
                   size_t inputDataSize,
                   double* output_buffer);
```

Description:
Computes the absorbance spectrum using a logarithmic transformation of transmission data based on Beer–Lambert law principles.

Parameters:

- spectrometerWavelength (const double*) – Wavelength array corresponding to each pixel.
- rawDarkDataY (const double*) – Dark spectrum measurement.
- rawRefeDataY (const double*) – Light reference (100% baseline) spectrum.
- rawSampleDataY (const double*) – Sample spectrum measurement.
- inputDataSize (size_t) – Total number of pixels in the input spectrum (i.e., the length of spectrometerWavelength)
- output_buffer (double*) – A preallocated array for the resulting absorbance values. **The buffer size must be at least as large as the length parameter.**

Returns:

- DevError status code (int)

---

loadCALFile

```
int loadCALFile(const char* path,
                double* calX,
                double* calY,
                size_t maxSize,
                size_t* calibrationSize,
                uint32_t* calibrationIntegrationTime);
```

Description:
Loads a .CAL file used for radiometric conversion. This file maps wavelengths to power for absolute intensity calibration.

Parameters:

- path (const char*) – Full file path to the .CAL file.
- calX (double*) – Output array buffer for calibration wavelengths. **The buffer size must be at least as large as maxSize.**
- calY (double*) – Output array buffer for calibration power values. **The buffer size must be at least as large as maxSize.**
- maxSize (size_t) – Expected maximum size of the provided buffers.
- calibrationSize (size_t*) – Returns the actual number of points loaded.
- calibrationIntegrationTime (uint32_t*) – Returns the integration time in milliseconds used during factory calibration.

Returns:

- DevError status code (int)

Usage Notes:

- Must be called before using getWattsY.
- The .CAL file must match the specific detector serial number.

getWattsY

```
int getWattsY(const double* spectrometerWavelength,
              const double* rawSampleDataY,
              const double* rawDarkDataY,
              size_t inputDataSize,
              uint32_t integrationTimeMs,
              const double* calX,
              const double* calY,
              size_t calSize,
              uint32_t calIntegrationTime,
              double aperturePercentage,
              double* output_buffer);
```

Description:
Converts raw spectral data into absolute optical power (Watts per wavelength). This function applies dark correction, calibration scaling, integration time normalization, and aperture correction.

Parameters:

- spectrometerWavelength (const double*) – Wavelength array corresponding to each pixel.
- rawSampleDataY (const double*) – Sample spectrum (raw)
- rawDarkDataY (const double*) – Dark spectrum
- inputDataSize (size_t) – Total number of pixels in the input spectrum (i.e., the length of spectrometerWavelength)
- integrationTimeMs (uint32_t) – Integration time, in milliseconds, used to acquire rawSampleDataY.
- calX (const double*) – Calibration arrays from **loadCALFile**.
- calY (const double*) – Calibration arrays from **loadCALFile**.
- calSize (size_t) – Calibration array size from **loadCALFile** (i.e. length of the calX and calY)
- calIntegrationTime (uint32_t) – Calibration integration time, in milliseconds, from **loadCALFile**.
- aperturePercentage (double) – Optical aperture scaling factor in percentage.
- output_buffer (double*) – A preallocated array to store the resulting absolute power spectrum. **The buffer size must be at least as large as inputDataSize.**

Returns:

- DevError status code (int)

Usage Notes:

- Requires valid .CAL calibration file.

---

getLumenY

```
int getLumenY(const double* spectrometerWavelength,
              const double* wattsDataY,
              size_t inputDataSize,
              double* output_wavelength,
              double* output_buffer,
              size_t* output_buffer_size);
```

Description:
Converts optical power (watts) into photometric units (lumens) over the 300–780 nm range with 5 nm increments.

Parameters:

- spectrometerWavelength (const double*) – Wavelength array corresponding to each pixel.
- wattsDataY (const double*) – Optical power spectrum from **getWattsY.**
- inputDataSize (size_t) – Total number of pixels in the input spectrum (i.e., the length of spectrometerWavelength)
- output_wavelength (double*) – A preallocated array to store the resulting wavelength scale for the lumen calculation. **Must contain space for at least 81 elements.**
- output_buffer (double*) – A preallocated array to store the calculated lumen values. **Must contain space for at least 81 elements.**
- output_buffer_size (size_t*) – Output size tracker. The expected value is 81.

Returns:

- DevError status code (int)

---

getParY

```
int getParY(const double* spectrometerWavelength,
            const double* wattsDataY,
            size_t inputDataSize,
            double* output_wavelength,
            double* output_buffer,
            size_t* output_buffer_size);
```

Description:
Computes Photosynthetically Active Radiation (PAR), representing light usable for plant photosynthesis, over the 300–1100 nm range with 1 nm increments.

Parameters:

- spectrometerWavelength (const double*) – Wavelength array corresponding to each pixel.
- wattsDataY (const double*) – Optical power spectrum from **getWattsY.**
- inputDataSize (size_t) – Total number of pixels in the input spectrum (i.e., the length of spectrometerWavelength).
- output_wavelength (double*) – A preallocated array to store the resulting wavelength scale. **Must be sized to accommodate at least 801 elements.**
- output_buffer (double*) – A preallocated array to store the calculated PAR values. **Must be sized to accommodate at least 801 elements**
- output_buffer_size (size_t*) – Output size tracker. The expected value is 801.

Returns:

- DevError status code (int)

interpolateLinear

```
int interpolateLinear(const double* inputX,
                      const double* inputY,
                      size_t inputLength,
                      double startX,
                      double endX,
                      double stepX,
                      double* outputX,
                      double* outputY,
                      size_t* outputLength);
```

Description:
Performs linear interpolation on spectral data to resample or normalize spectra across a user-defined wavelength range and interval. This function is essential for transforming raw, irregularly spaced pixel data into a standardized format.

Parameters:

- inputX (const double*) – The original wavelength array (independent variable).
- inputY (const double*) – The original intensity or power array (dependent variable).
- inputLength (size_t) – The total number of points in the input arrays.
- startX (double) – The desired starting wavelength for the interpolated range.
- endX (double) – The desired ending wavelength for the interpolated range.
- stepX (double) – The fixed wavelength increment between points (e.g., 1.0 nm).
- outputX (double*) – Preallocated array to store the new, evenly spaced wavelength values.
- outputY (double*) – Preallocated array to store the calculated intensity values at the new wavelengths.
- outputLength (size_t*) – Pointer to an integer that will be updated with the actual number of points generated in the output arrays.

Returns:

- DevError status code (int)

Usage Notes:

- **Alignment & Normalization:** Highly useful for aligning spectra from different spectrometers or matching experimental data to theoretical models and standards.
- **Data Consistency:** The **inputX** array must be **monotonic** (strictly increasing or decreasing) for the interpolation logic to function correctly.

Version Control

get_version

```
const char* get_version(void);
```

Description:
Returns the current version string of the library. This is used to verify SDK compatibility, track software builds, and support debugging across deployments.

Returns:

- const char* – Null-terminated version string

Usage Notes:

- Useful for logging and compatibility checks
- Helps ensure correct driver library is loaded in multi-version environments

Version Control

This section tracks the version history of the StellarNet SDK and documentation. It allows users to identify the release they are using and follow updates, bug fixes, and feature additions over time.

Contact / Support

For questions, feedback, suggestions, or bug reports regarding the StellarNet SDK, please contact the StellarNet software team at:

Email: software@stellarnet.us

When reporting issues or bugs, please include as much relevant detail as possible (e.g., system setup, programming language, code snippets, and error messages). If you have identified a potential cause or workaround, feel free to include that information as well, as it may help speed up resolution.

We welcome contributions from users. If you have developed example projects, integrations, or demos in other programming languages or applications using the StellarNet SDK that you would like to share, please feel free to share them with us. Your contributions may help improve the SDK and support other developers in the community.