

NSP32 SDK[®]

C/C++ Developer Guide

ver 1.7

nanoLambda

IMPORTANT NOTICE

nanoLambda Korea and its affiliates (“nanoLambda”) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to nanoLambda’s terms and conditions of sale supplied at the time of order acknowledgment. Customers are responsible for their products and applications using any nanoLambda products. nanoLambda does not warrant or represent that any license, either express or implied, is granted under any nanoLambda patent right, copyright, mask work right, or other nanoLambda intellectual property right relating to any combination, machine, or process in which nanoLambda products or services are used. Information published by nanoLambda regarding third-party products or services does not constitute a license from nanoLambda to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from nanoLambda under the patents or other intellectual property of nanoLambda. Reproduction of nanoLambda information in nanoLambda documents or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. nanoLambda is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions. Resale of nanoLambda products is not allowed without written agreement. Decompiling, disassembling, reverse engineering or attempt to reconstruct, identify or discover any source code, underlying ideas, techniques or algorithms are not allowed by any means. nanoLambda products are not authorized for use in safety-critical applications. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of nanoLambda products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by nanoLambda. Further, buyers must fully indemnify nanoLambda and its representatives against any damages arising out of the use of nanoLambda products in such safety-critical applications. This Notice shall be governed by and construed in accordance with the laws of Korea, without reference to principles of conflict of laws or choice of laws. All controversies and disputes arising out of or relating to this Notice shall be submitted to the exclusive jurisdiction of the Daejeon District Court in Korea as the court of first instance.

Application Programming Interface (API) For NSP32 Spectral Sensor

NSP32 application programming interface (API) is a list of all classes that are part of the NSP32 Software Development Kit (SDK). It includes all libraries, classes, and interfaces, along with their methods, fields, and constructors. These prewritten classes provide a tremendous amount of functionality to a programmer. A programmer should be aware of these classes and should know how to use them. If you browse through the list of packages in the API, you will observe that there are packages written for reading data from NSP32 spectral sensor, connecting single or multiple sensors, managing input and output, getting spectrum data for single or multiple sensors, and many more. Please browse this manual for complete list of available functions and their descriptions to see how they can be used.

Table of Contents

ABOUT THIS MANUAL	6
CHAPTER 1 INTRODUCTION	8
OVERVIEW	8
NSP32 SDK IS CROSS-PLATFORM	10
NSP32 SDK IS WRITTEN IN C/C++	10
NSP32 SDK ARCHITECTURE	12
USB PORT ACCESS	12
RELATED MANUALS YOU MUST READ	12
HOW TO USE THIS DOCUMENT	12
CHAPTER 2 NSP32 SDK INSTALLATION	13
OVERVIEW	13
RETRIEVING FROM A USB STICK	13
INSTALLING ON A WINDOWS PLATFORM	14
INSTALLING ON A MAC PLATFORM	15
INSTALLING ON A UBUNTU PLATFORM	16
INSTALLING THE USB DRIVER SOFTWARE	19
CHAPTER 3 BEFORE START TO DEVELOP	20
OVERVIEW	20
PHASE I: EVALUATION	20
PHASE II: APPLICATION DEVELOPMENT	21
CHAPTER 4 BASIC SEQUENCE OF OPERATIONS	24
OVERVIEW	24
WITH HIGH-LEVEL API	25
<i>Create High-Level API Object</i>	<i>26</i>
<i>Activate One NSP32 Spectral Sensor</i>	<i>26</i>
<i>Set Sensor Parameters</i>	<i>27</i>
<i>Get a Spectrum</i>	<i>28</i>
<i>Destroy High-Level API Object</i>	<i>28</i>
WITH LOW-LEVEL API	29
<i>Connect and Identify NSP32 Spectral Sensor</i>	<i>30</i>
<i>Create Core Spectrum Object</i>	<i>31</i>
<i>Get/Set Sensor Parameters</i>	<i>31</i>
<i>Activate Sensor and Cal Data</i>	<i>32</i>
<i>Get/Set Background Data</i>	<i>32</i>

<i>Get/Set Optimal Shutter Speed</i>	33
<i>Get Raw Sensor Data and Calculate Spectra</i>	33
<i>Destroy Core Spectrum Object</i>	34
<i>Disconnect NSP32 Spectral Sensor</i>	34
CHAPTER 5 ACQUISITION PARAMETERS	35
OVERVIEW	35
SHUTTER SPEED	35
AUTO-EXPOSURE	36
ADC SETTINGS(ADC GAIN & RANGE).....	37
FRAME COUNT FOR AVERAGING	38
CHAPTER 6 HIGH-LEVEL API	39
OVERVIEW	39
CREATE SENSOR INTERFACE	39
ADD A CALIBRATION DATA TO DATA REPOSITORY.....	40
ACTIVATE A NSP32 SPECTRAL SENSOR.....	40
GET ACTIVATED SENSOR INFORMATION	41
GET/SET SENSOR PARAMETERS	41
FIND OPTIMAL SHUTTER SPEED.....	42
SHUTTER SPEED TO EXPOSURE TIME.....	43
ACQUIRE A SPECTRAL.....	43
DESTROY SENSOR INTERFACE	44
CHAPTER 7 LOW-LEVEL API	45
OVERVIEW	45
CONNECT NSP32 SPECTRAL SENSOR.....	45
ACTIVATE PHYSICAL SENSOR.....	45
GET TOTAL SENSORS IN SYSTEM.....	46
GET MAXIMUM COUNT OF SENSORS.....	46
GET SENSOR LIST IN SYSTEM.....	46
GET PHYSICAL SENSOR ID	47
GET A LIST OF CONNECTED SENSOR IDS	47
SHUTTER SPEED TO EXPOSURE TIME/	47
EXPOSURE TIME TO SHUTTER SPEED.....	47
GET OPTIMAL SHUTTER SPEED	48
GET FILTER DATA	49
GET VALID FILTERS	50
GET/SET SENSOR PARAMETERS	50
DISCONNECT NSP32 SPECTRAL SENSOR.....	51

CREATE CORE SPECTRUM OBJECT	52
REGISTER A CALIBRATION DATA TO REPOSITORY	52
ACTIVATE CALIBRATION SENSOR DATA.....	53
GET SPECTRAL INFORMATION	53
GET SENSOR ID FROM CALIBRATION FILE	54
GET CAPACITY AND SENSOR LIST FROM REPOSITORY	54
GET SENSOR PARAMETERS FROM CAL FILE.....	55
SET BACKGROUND FILTER DATA	55
ACQUIRE A SPECTRAL	56
DESTROY SENSOR INTERFACE	57
CHAPTER 8 DEVELOPING YOUR NSP32 SDK APPLICATION	58
OVERVIEW	58
ANDROID STUDIO SETUP ON LINUX.....	58
DEVELOPING IN C/C++ (ALL IDEs)	59
DEVELOPING WITH MICROSOFT VISUAL C++ 10.0.....	60
MODIFYING YOUR C++ PROJECT SETTINGS FOR 64-BIT WINDOWS	63
NETBEANS SETUP ON LINUX.....	64
DEVELOPING IN MATLAB.....	66
DEVELOPING IN LABVIEW.....	68
CHAPTER 9 EXAMPLE PROGRAMS.....	71
OVERVIEW	71
C/C++ EXAMPLES.....	71
MATLAB EXAMPLES	72
LABVIEW EXAMPLES.....	73
PYTHON EXAMPLES	73
APPENDIX A HOW TO VALIDATE YOUR RAW SENSOR DATA?	75
APPENDIX B USB DRIVER INSTALLATION.....	79
INSTALLING THE USB DRIVER ON WINDOWS 7/8/10	79
INSTALLING THE USB DRIVER ON ANDROID, LINUX, MACOS, AND RASPBIAN	80
APPENDIX C NSP32 SDK EXAMPLES (HIGH-LEVEL API)	81
APPENDIX D NSP32 SDK EXAMPLES (LOW-LEVEL API).....	84
APPENDIX E NSP32 SDK EXAMPLES (MATLAB WRAPPER).....	88
APPENDIX F NSP32 SDK EXAMPLES (PYTHON WRAPPER).....	90

About This Manual

Document Purpose and Intended User

This document provides you with an installation and development instructions and function references for NSP32 SDK.

What's New in this Manual

This version of the *NSP32 SDK Developer Guide* is updated for Release 1.7.

Document Summary

Chapter	Description
Chapter 1: <i>Introduction</i>	Provides an overview of the NSP32 SDK software.
Chapter 2: <i>NSP32 SDK Installation</i>	Contains instructions for installing NSP32 SDK on a Windows, Mac or Linux platform from the NSP32 SDK Installation Package.
Chapter 3: <i>Before Start to Develop</i>	Describes the 3 typical use cases of NSP32 SDK to control NSP and to have and/or nano-filter data and spectral data.
Chapter 4: <i>Basic Sequence of Operations</i>	Describes the typical sequence of operations that the application must perform to control a nanoLambda's NSP32 spectral sensor and acquire nano-filter/spectral data.
Chapter 5: <i>Acquisition Parameters</i>	Presents a complete list of all available acquisition parameters.
Chapter 6: <i>High-Level API</i>	Lists and describes the High-Level functions to control NSP32 and to acquire spectral data.
Chapter 7: <i>Low-Level API</i>	Lists and describes the Low-Level functions to control NSP32 and to acquire nano-filter data and spectral data.
Chapter 8: <i>Developing Your NSP32 SDK Application</i>	Provides information on how to set up the development environment for user specific programming language to develop NSP32 SDK

	applications.
Chapter 9: <i>Example Programs</i>	Contains example programs for C/C++, Matlab, Python, and LabVIEW.
Appendix A: <i>How to Validate Your Raw Sensor Data?</i>	Provides the detailed description how to validate your own sensor data from NSP32.
Appendix B: <i>NSP32-USB Driver Installation</i>	Provides the detailed instructions to install USB driver for NSP32 spectral sensor.
Appendix C: <i>NSP32 SDK Examples (High-Level API)</i>	Provides a representative C/C++ example source code in NSP32 SDK (High-Level API).
Appendix D: <i>NSP32 SDK Examples (Low-Level API)</i>	Provides a representative C/C++ example source code in NSP32 SDK (High-Level API).
Appendix E: <i>NSP32 SDK Examples (MATLAB Wrapper)</i>	Provides a representative Matlab example source codes in NSP32 SDK.
Appendix F: <i>NSP32 SDK Examples (Python Wrapper)</i>	Provides a representative Python example source codes in NSP32 SDK.

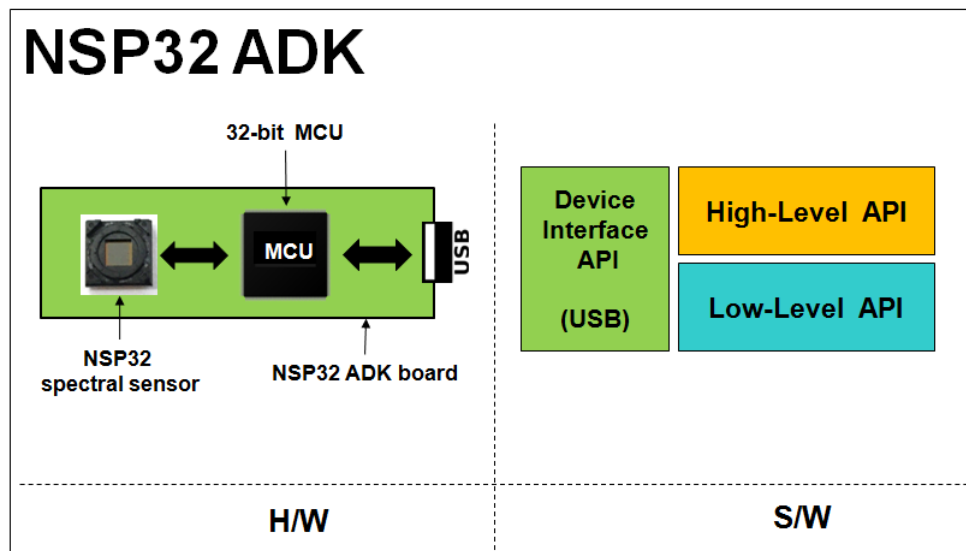
List of Acronyms

NSP32	Model name for a nano spectrometer
NSP32 ADK	Application Development Kit with NSP32 sensor and SDK
NSP32 SDK	Software Development Kit for NSP32 sensor
Low-Level API	Low-Level API in NSP32 SDK
High-Level API	High-Level API in NSP32 SDK

Chapter 1 Introduction

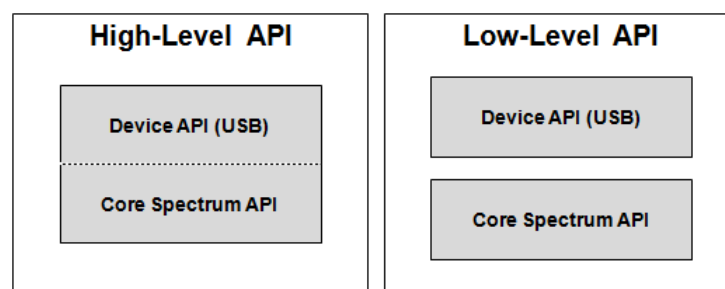
Overview

NSP32 SDK is a Software Developer's Kit (SDK) for Android, Windows, Mac OS and Linux (including variants like Raspbeian, Angstrom, and Debian) operating systems that allows developer to easily write custom software solutions for your NSP32 (nanoLambda's spectral sensor).



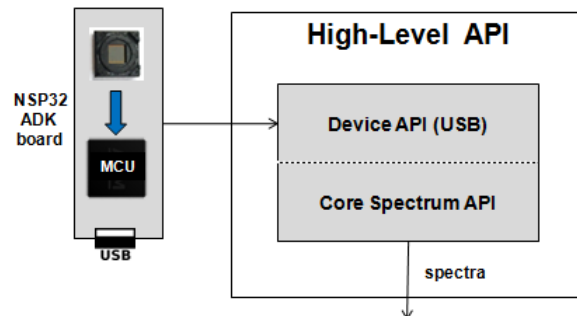
NSP32 SDK provides 2 different levels of Application Programming Interface (API) written with C/C++ and allows user to harness the applicability of miniaturized NSP32 spectral sensor for a variable spectral sensing applications. You can develop your software and even hardware application with APIs in NSP32 SDK and NSP32 spectral sensor in any major OS environments (Windows, Mac OS and Linux).

NSP32 SDK includes 2 different levels of API:

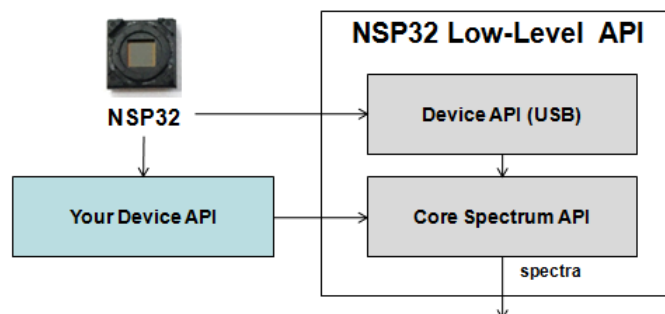


- **High-Level API.** This API provides a high-level API for USB-based Device Interface and Core Spectrum Interface functions. This is less flexible than the Low-Level API, but it relieves you of the burden of custom development of your hardware interface. With this

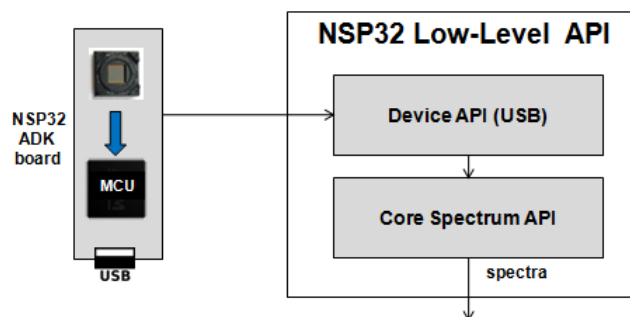
highly abstracted API, user will be free from to know and control every details of device (NSP32) interface, NSP32's register settings, nano-filter data acquisition, and spectra calculation from nano-filter data. This allows you to prototype your spectral sensing application and verify its commercial viability more quickly with NSP32 ADK board.



- **Low-Level API.** This API provides more flexible way to develop your own spectral sensing application by providing the separate interface to the Device Interface object and the Core Spectrum Interface object. That is, if you want to develop your own sensor interface to control NSP32 spectral sensor by yourself, then you don't need to use DeviceInterface (USB-based) in this Low-Level API. With your own device interface, you can acquire a sensor data from NSP32 spectral sensor and then can calculate your spectra by using Core Spectrum Interface.



You can also develop your application with ADK board and Low-Level API.



You can use a specific combination of NSP32 ADK, NSP32 spectral sensor and API for your application development:

- NSP32 ADK + High-Level API
- NSP32 ADK + Low-Level API
- NSP32 spectral sensor + Low-Level API

NSP32 SDK is Cross-platform

NSP32 SDK was created in the C/C++ environment and supports Android, Raspberry-Pi, Windows, Macintosh and Linux operating systems by cross-compiled libraries of C/C++. Using NSP32 SDK, you can develop spectral sensing applications with nanoLambda's NSP32 spectral sensor across these different operating systems.

Multi-Languages Support

You can develop NSP32 ADK-based applications in the following languages.

- C (with Microsoft Visual Studio/Windows, also with gcc/Mac OS and Linux)
- C++ (with Microsoft Visual Studio/Windows, also with g++/Mac OS and Linux)
- Matlab (2013a or higher)
- Python (Version 2.xx and 3.xx)
- LabVIEW (Windows only, Version 13 or greater)

NSP32 SDK is Written in C/C++

All of NSP32 SDK's capabilities are implemented in the C/C++ language. This is key to NSP32 SDK's ability of easy porting to multiple platforms. As a consequence, C/C++ developers can easily learn and use the C/C++ objects and functions in static/shared libraries directly. And for other languages like Matlab, Python, and LabVIEW, NSP32 SDK provides language specific Wrappers to developers.

NSP32 SDK Features

NSP32 SDK offers the following features:

Operating System Support

- Android:
 - Lollipop (5.0) or higher version
- Windows:
 - Windows 7 (32-bit and 64-bit versions)
 - Windows 8, 8.1, and 10 (64-bit versions)
- Mac OS:
 - OS X 10.10 (Yosemite) or later
- Linux (Ubuntu):
 - Many x86 distributions are supported
 - Kernel 3.13 or higher (Ubuntu, 14.04 or higher).
 - 3rd Party libraries: libgsl and libusb 1.0.9 are required
- Raspbian:
 - Raspberry-Pi 2 (Model B) and 3 (Model B) are supported
 - Kernel 4.4
- Anstrom/Debian Distribution:

- BeagleBone-Black is supported

MATLAB, Python and LabVIEW Support

NSP32 SDK provides API Wrappers for MATLAB(2013a or higher), Python(2.xx and 3.xx) and LabVIEW (version 13 or higher) to enable you to connect nanoLambda's NSP32 spectral sensor in MathWorks' MATLAB software environment, Python Software Foundation's Python programming language, and National Instruments' LabVIEW graphical programming environment.

Table 1. NSP32 supports multiple programming languages and platforms

	Android	Windows	Mac OS	Linux	Raspbian	Angstrom /Debian
C/C++	●	●	●	●	●	●
Matlab	-	●	-	-	-	-
Python	-	●	-	●	●	-
LabVIEW	-	●	-	-	-	-

Note: Multiple-languages support

Although NSP32 SDK supports different programming languages, this Developer Guide is written for C/C++ developers. If you want to use MATLAB, Python or LabVIEW for your application development, you need to read other documents additionally. You can find those documents for language Wrappers under '/doc' folder:

- MATLAB: '*NSP32 SDK Reference Manual for Matlab-v1.7.pdf*'
- Python: '*NSP32 SDK Reference Manual for Python-v1.7.pdf*'
- LabVIEW: '*NSP32 SDK Reference Manual for LabVIEW-v1.7.pdf*'

MATLAB Development

For MATLAB developers, NSP32 SDK provides a set of wrapper functions which expose NSP32 SDK's functionality in *.m files for Windows OSs. These wrapper M files invoke the methods contained in the DLL files that comprise NSP32 SDK for Windows.

Python Development

For Python developers, NSP32 SDK provides a set of wrapper functions which expose NSP32 SDK's functionality in *.py files for Windows, Linux, and Raspbian OSs. These wrapper py files invoke the methods contained in the shared file that comprise NSP32 SDK.

LabVIEW Development

For LabVIEW developers, NSP32 SDK provides a set of VI files which expose NSP32 SDK's functionality in a fashion that is natural to the LabVIEW development environment. These VI's invoke the methods contained in the DLL files that comprise NSP32 SDK for Windows.

NSP32 SDK Architecture

You can access NSP32 SDK functionality via 3 different sets of files:

- **CrystalBase.dll, CrystalCore.dll, CrystalPort.dll** (32/64-bit versions) contains the Low-Level API functions which allow you to control all NSP32 sensor settings and acquire spectra through USB connection. For example, you can access NSP32 spectral sensor's registers, set integration time, specify frames-to-average, get spectra, etc.
- **CrystalBox.dll** (32/64-bit version) contains High-Level API functions for connecting NSP32 sensor, setting sensor parameters, and calculating the spectra.

USB Port Access

The first product of nanoLambda, NSP32 spectral sensor on NSP32 ADK board, can communicate with a computer via the USB port. To access the USB ports, we provide a DLL named CrystalPort.dll. The methods in this DLL are invoked internally from the High-Level and Low-Level APIs of NSP32 SDK. For other communication schemes like Ethernet or BlueTooth, you have to implement custom device interface (driver) by yourself.

Related Manuals You Must Read

- NSP32 Data Sheet
- NSP32 SDK Installation Manuals
- NSP32 SDK API Reference Manuals

How To Use This Document

You can use a specific combination of NSP32 ADK, NSP32 spectral sensor and API for your application development. nanoLambda suggests to read chapters for each combination in this document:

- **NSP32 ADK + High-Level API**
 - Chapter 3, Chapter 4, Chapter 5, Chapter 6, and Chapter 9
 - Appendix C (Example)
- **NSP32 ADK + Low-Level API**
 - Chapter 3, Chapter 4, Chapter 5, Chapter 7, and Chapter 9
 - Appendix D (Example)
- **NSP32 spectral sensor + Low-Level API**
 - Chapter 3, Chapter 4, Chapter 5, Chapter 7, and Chapter 9
 - Appendix A, Appendix D (Example)

Chapter 2 NSP32 SDK Installation

Overview

NSP32 SDK can be retrieved from the SDK Installation Package in a USB stick that you received with your purchase of NSP32 spectral sensor or from Cloud Server.

This chapter contains instructions for installing NSP32 SDK on each of the following operating systems:

- **Android** – Lollipop (5.0) or higher
- **Microsoft Windows** – 7, 8, 8.1, 10; 32-bit and 64-bit
- **Mac OS** – OS X version 10.10 or later on Intel processor
- **Linux(Ubuntu)** – 14.04 (Trust Tahr) or higher version

Retrieving from a USB Stick

Your NSP32 SDK software may be shipped to you from nanoLambda on a separate USB Stick labeled NSP32 SDK.

► Procedure

1. Insert the USB Stick that you received containing your NSP32 SDK software into your computer.
2. Select the NSP32 SDK installation package file for your computer's operating platform. Then follow the instructions for SDK installation procedure (see sub-sections).

`NSP_SDK_1_7_package.tar.gz`

3. Refer to the appropriate installation section, depending on your operating platform to complete the installation. All API files and sample applications are now located under a separate folder for different OS platforms.
4. Install USB driver to build USB connection to NSP32 spectral sensor (please refer to "*Appendix C: USB Driver Installation*" section).

When the installation process is finished, the following subdirectories will be created beneath the NSP32 SDK "home" directory:

Subdirectory	Contents
docs	Main documentation area (PDF files)
include\	Header files for use with C/C++ application development

lib\android	Static library files for Android
lib\macOS	Shared library files for MacOS
lib\raspbian-jessie	Static library files for Raspbian-Jessie
lib\ubuntu	Static library files for Ubuntu
lib\win32	Static and shared library files for Windows 32-bit
lib\win64	Static and shared library files for Windows 64-bit
bin\android	Sample executables for Android
bin\macOS	Sample executables for MacOS
bin\raspbian-jessie	Sample executables for Raspbian-Jessie
bin\ubuntu	Sample executables for Ubuntu
bin\win32	Sample executables for Windows 32-bit
bin\win64	Sample executables for Windows 64-bit
examples\matlab	Example files for MATLAB
examples\labview	Example files for LabVIEW
examples\cpp	Example files for C/C++
examples\python	Wrappers(*.py) and example files for use with Python development environment.
wrappers\labview	Wrappers(*.vi) for use with LabVIEW development environment
wrappers\matlab	Wrappers(*.m) for MATLAB development environment.
wrappers\python	Wrappers(*.py) for Python development environment.
prism_gui\win32	Prism GUI application
tools	Tools for USB driver installation and for validating raw filter data

Once you have installed the software, you'll want to verify your installation. To verify the installation, please look at the documents provided to get an idea of how the objects and methods for NSP32 SDK are organized, and then run a sample program.

Installing on a Windows Platform

Installing .NET Framework

Before you can install the NSP32 ADK software, you must ensure that the .NET framework has

been installed. NSP32 SDK requires version 4.0 or later. You can download the installer for .NET from Microsoft's website (<https://www.microsoft.com/net/download/framework>).

► Procedure

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

Installing NSP32 SDK on a Windows Platform

If your operating system is Windows (7, 8, 8.1, or 10), then you need to run installation program.

► Procedure

1. Double click '**NSP_SDK_version_number_installer_win32.exe**' or '**NSP_SDK_version_number_installer_win64.exe**' depends on the your platform and install feature.
During the installation you may be prompted with a message shown below. If so, use the following procedure.

<figures for installation>....

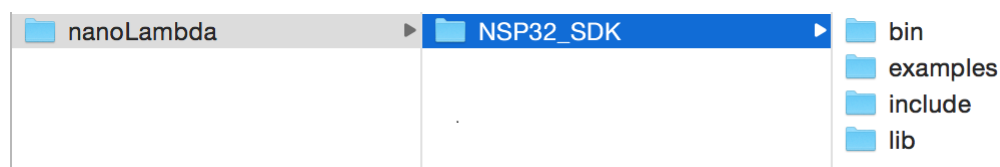
2. After successfully completing the installation wizard, plug in your USB connected NSP32 spectral sensor included in your ADK and follow the procedure for installing the necessary USB driver for your sensor (refer to a PDF document under '**/doc/NSP32 USB Driver Installation Manual-v1.7.pdf**').
3. Once you have installed the SDK software and USB driver, you'll want to verify your installation. To verify the installation, please run a sample program (for example, '[NSP32_SDK_HOME]/bin/win32/cpp/examples_spectrum_api.exe').

Installing on a Mac Platform

To install NSP32 SDK on Mac, run a command from your local directory:

```
$ tar xvf ./nsp32_sdk_for_Mac.tar
```

After untar the file, you can find below paths in that path.



Here are the contents.

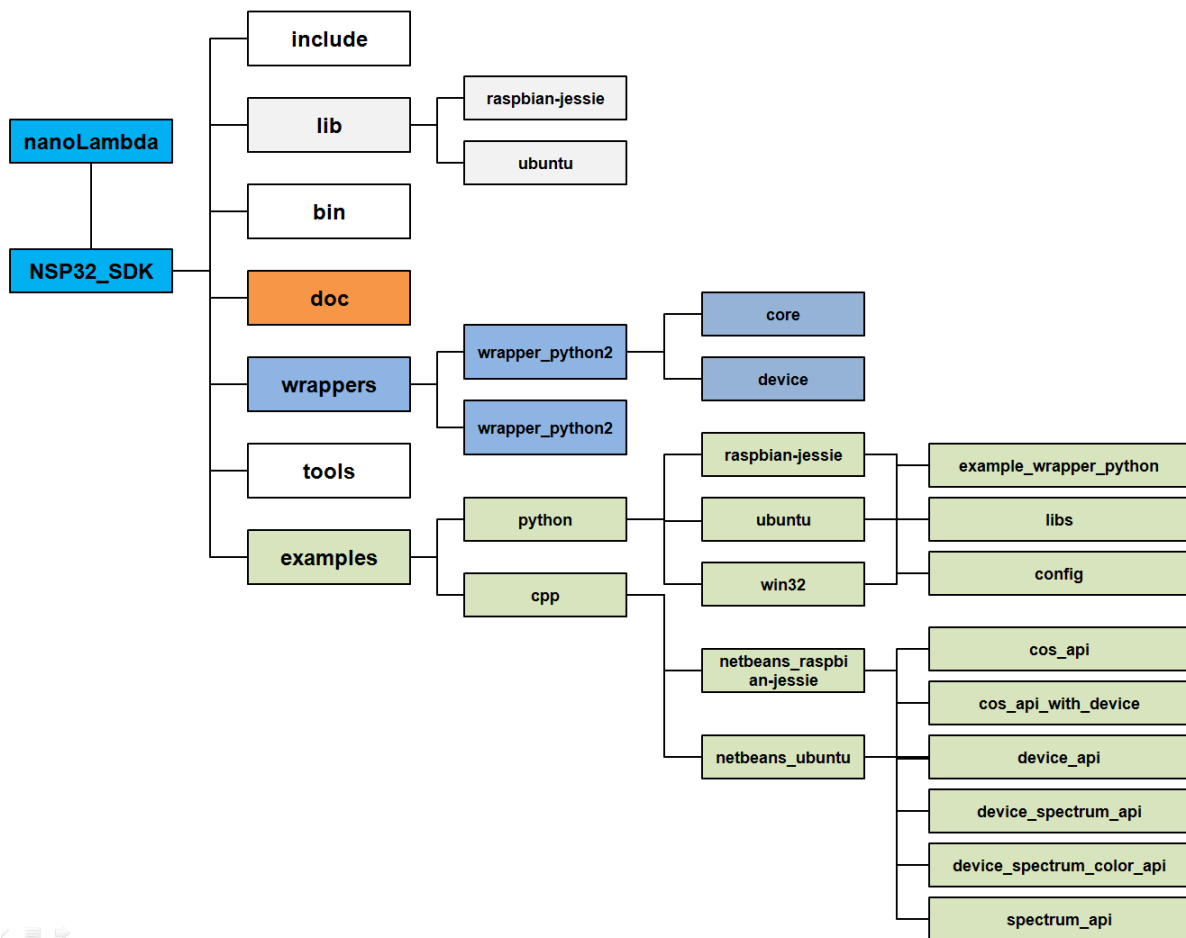
- **bin** : PrismEx.dmg - GUI application for NSP32 sensor
- **examples** : sample example for using NSP32 sensor
- **include** : header files
- **lib** : all libraries

Installing on a Ubuntu Platform

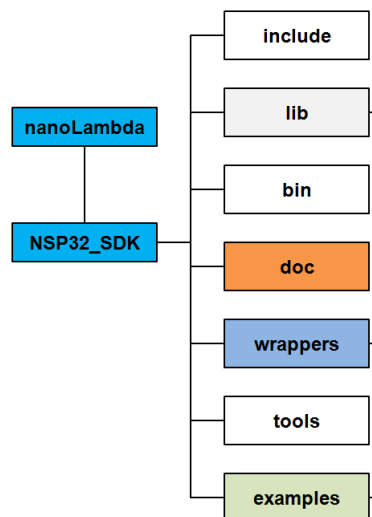
We provide a convenient installation file for NSP32 SDK on the Linux family of operating systems ('NSP32_SDK_1_7_installer_ubuntu.tar.gz'). This package is compressed in .tar.gz format. To install NSP32 SDK on Ubuntu, double click this file or you can uncompress it by using the command below:

```
$ tar xvf NSP32_SDK_1_7_installer_ubuntu.tar.gz
```

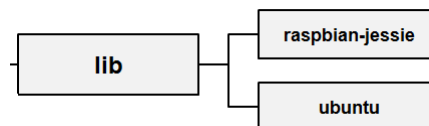
After installation, You will get the directory structures as below:



NSP32_SDK folder contains sub-directories.

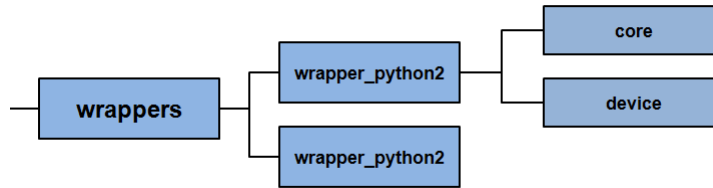


- "include" folder contains all the header files, you need to run the example code.
- "lib" folder contains all the static libraries. "lib" folder has sub-directories which in-turn has all libraries for specific platforms like Raspbian-jessie and Ubuntu OSs.

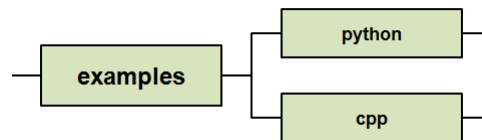


- "doc" folder contains all the documents. Right now there are 3 manuals (PDF formats). Two are manuals for installation and running example codes for RaspberryPi and Ubuntu, respectively. The other one is Python API reference manual.
 - *"NSP32 SDK Installation Manual for Ubuntu-v1.7.pdf"*
 - *"NSP32 SDK Installation Manual for RaspberryPi-v1.7.pdf"*
 - *"NSP32 SDK Reference Manual for Python-v1.7.pdf"*
- "wrappers" folder contains wrapper for python. nanoLambda NSP32 SDK supports Python language with Python wrappers for both version of Python 2.xx and 3.xx.
- Under 'NSP32_SDK/wrappers/Python' folder, you can find two separate wrapper folders, "wrapper_Python2" and "wrapper_Python3". In each folder, there are 2 sub-directories, "core" and "device". "core" folder includes all the functions related to the core spectrum of the spectral sensor and "device" folder includes all the functions related to NSP32 device interface. All the codes are written in C/C++ and they

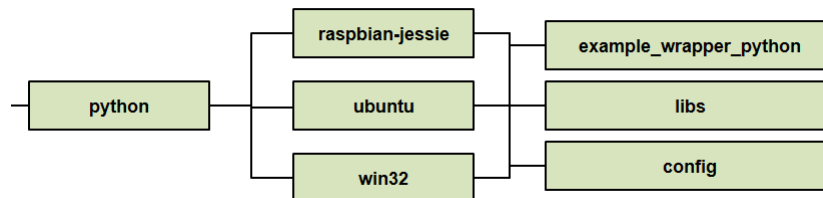
are just wrapped with an additional layer to provide Python interface.



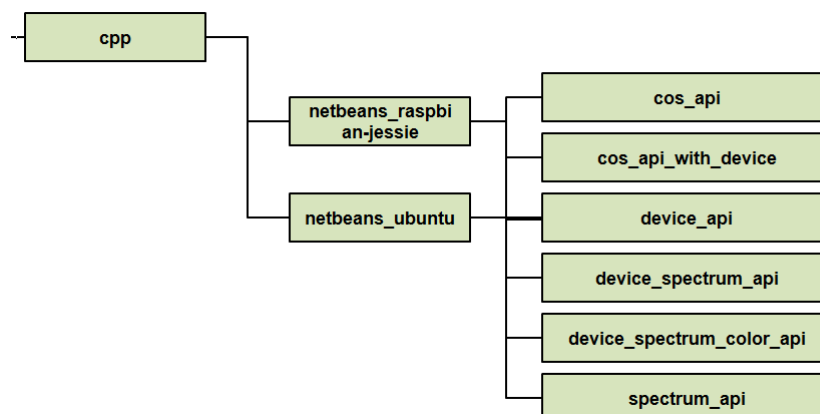
- "examples" folder contains examples for "cpp" and "python" languages.



- "python" folder contains 3 sub-directories, platform specific. "raspbian-jessie" for Raspberry-pi, "ubuntu" for Ubuntu and "win32" for Windows 32-bit version. Each folder also has 3 sub-folders. "example_wrapper_python" folder contains an example code which tells the usage of almost all the wrapper functions. "libs" folder contains the library for platform specific. For getting the accurate spectral data from the NSP32 spectral sensor, user need to put sensor specific sensor data file in "config" folder.



- "cpp" folder contains two folders, one for Raspbian-jessie and one for Ubuntu. Each folder has 6 more directories to show how user can get the data and other information from NSP32 spectral sensor.



Installing the USB Driver Software

Please refer to please refer to '*Appendix C: USB Driver Installation Guide*' section.

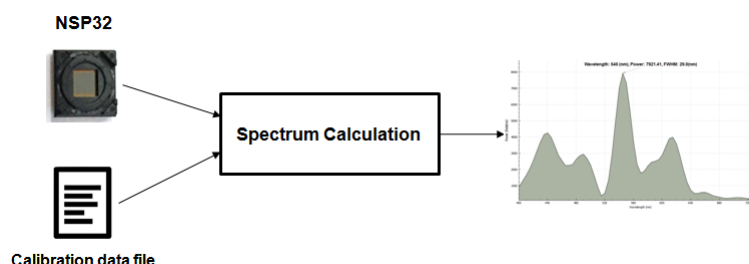
Chapter 3 Before Start To Develop

Overview

The process of obtaining spectra from nanoLambda's NSP32 spectral sensor is different from other commercially available spectrometers. The approach by nanoLambda is as follows:

1. Obtain nano filter data from NSP32 spectral sensor or NSP32 ADK board
2. Calculate spectrum using spectral sensor calibration data and nano filter data

Acquisition of nano-filter data, loading of sensor calibration data, and calculation of spectra are all performed on your processor. NSP32 SDK provides API functions to acquire nano-filter data, to load sensor calibration data, and to calculate spectra with those two data:



The application development steps with nanoLambda's NSP32 ADK, NSP32 SDK and NSP32 spectral sensor can be divided into three steps as follows:

- The applicability evaluation of nanoLambda's NSP32 ADK and APIs
- The development of your own application with NSP32 SDK and NSP32 spectral sensor and performance evaluation
- Market launch of application products.

In this chapter, we will focus the discussion on steps 1) and 2).

Phase I: Evaluation

You can use the nanoLambda's NSP32 ADK to evaluate the performance of the nanoLambda NSP32 spectral sensor to determine what is appropriate for your particular application, what limits are imposed on it, what parts need to be improved for commercial applications (S / N ratio, Dynamic range, sensitivity, stability, repeatability, accuracy, spectral resolution, etc.). Performance and applicability assessments can be performed using the Prism GUI application that can run on the OSs platform. This is easier and time-saving way. To do this, you need to use typically nanoLambda's NSP32 ADK in which sensor data acquisition board (NSP32 spectral sensor + MCU on PCB), USB connector, APIs, and GUI application are contained. The following schematic shows an example of an evaluation procedure using nanoLambda's NSP32 ADK.

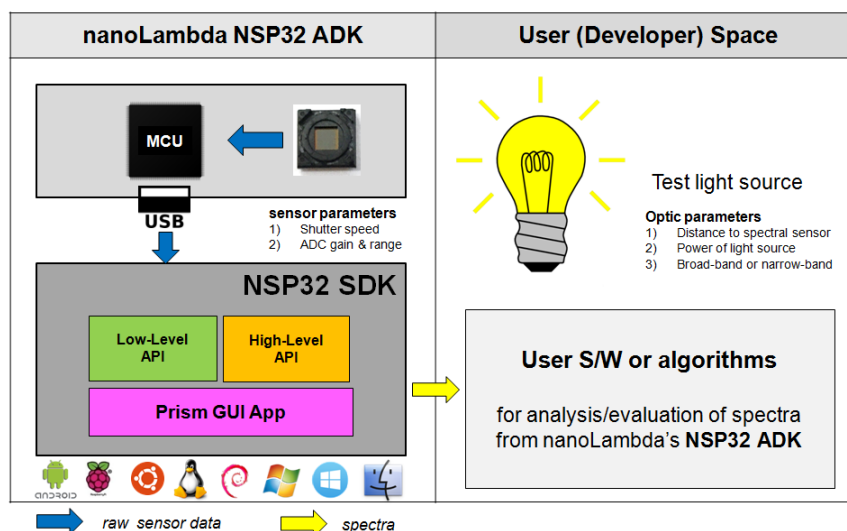


Figure. 1 NSP32 ADK based evaluation configuration and procedure.

More specifically, the Prism GUI application provides integrated connectivity to NSP32 spectral sensor, easy configuration of sensor parameters, loading of sensor-specific calibration data matched to the physical ID of NSP32 spectral sensor, and functions related to the calculation of the spectra. You can also use the Prism GUI

- to determine optimal or semi-optimal sensor parameters (especially shutter speed or exposure time, frame count for averaging)
- to find optimal optical setup (narrow-band, broad-band or combined light source, distance to NSP32 spectral sensor, etc.).

Once the sensor parameters and the optical setup for the acquisition of reliable spectra are established, you can use the Prism GUI application to acquire spectra repeatedly or under different conditions and store them in a text format file (CSV format). The spectra data stored in the text file can be used for analysis/evaluation using your own tools or algorithms and the result can be used to determine if the nanoLambda NSP32 spectral sensor can be applied to your application.

Phase II: Application Development

If you have decided to develop your own application using NSP32 spectral sensors based on the evaluation results with nanoLambda's NSP32 ADK, you can now proceed to the next step: *how to use nanoLambda's NSP32 spectral sensor and software APIs to your application?*

The goal of NSP32 spectral sensor and software APIs from your point of view is very clear: *spectra acquisition for your application*. In this second phase, you will be more cautious than the evaluation phase with the NSP32 ADK because you have to be consider more realistically about the footprint of your application, the user interface, the usage environment, the time, cost, and other.

In this chapter, we are going to help you make your decision through three possible ways you can refer to how to integrate nanoLambda's NSP32 spectral sensor into your application. If your application has a unique development approach, the descriptions of the three APIs presented in later chapters of this document will give you more specific and practical help. For example, depending on the development method you choose, you may need to consult only a chapter on the High-Level API, or you may need to write application code based on a Low-Level API.

There are two ways in which you can choose:

1. Integrate the ADK itself as part of your application

Use Case

Obtain spectrum data using nanoLambda's High-Level API on nanoLambda's ADK 'compatible' platform (e.g. Windows platform plus USB connection).

The way you integrate nanoLambda's ADK itself into your system is appealing because it can save you a lot of time and effort. The prerequisite is that your application system / platform must be compatible with the ADK. Your system must have a USB port and the OS platforms must be Android, Windows, Mac, Ubuntu, Debian (for Beagle-Bone), and Raspbian (for Raspberry). The APIs are written in C / C++ and provide wrappers for MATLAB, LABView, and Python.

If your application system meets these requirements, you can quickly acquire the spectral data you want using NSP32 spectral sensor and Low-Level or High-Level APIs provided by nanoLambda's NSP32 SDK. This method includes choosing the type of light source that ensures maximum extraction of spectral fingerprints that are specific to your application and selecting the optimal sensor parameters (shutter speed or exposure time) to obtain stable and reliable spectra in a configured application environment, and developing algorithms that process / analyze a set of acquired spectral data or spectra for your application purpose.

However, the disadvantage of this method lies in the size and price of the ADK. If you want to develop a miniaturized, monolithic, stand-alone device (such as a real-time CCT monitoring device) for your application and ship it to the market in large quantities, then ADK-based application development might not be a solution. You must go to method 2) or 3) below.

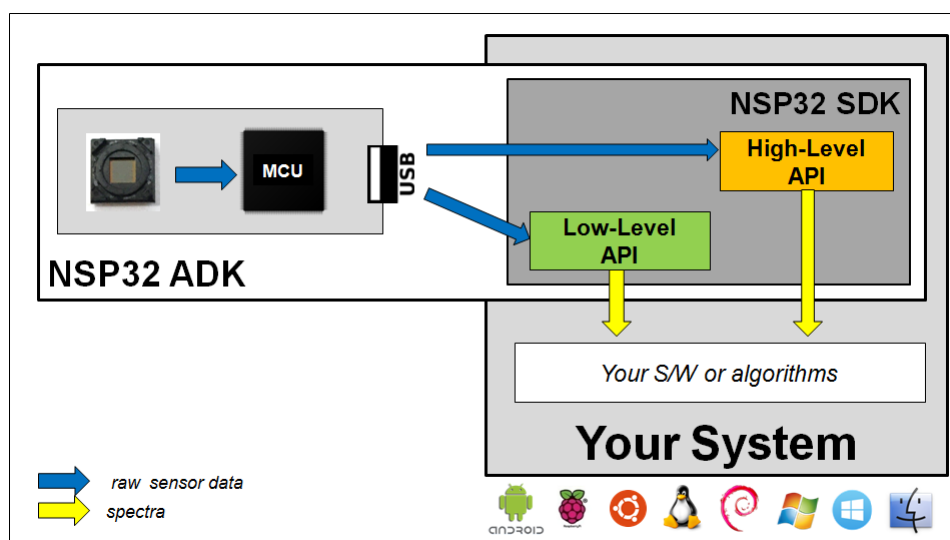


Figure.2 Integration of NSP32 ADK into the part of your system directly.

2. The hardware interface for NSP32 spectral sensor is developed by you and you select an OS platform that is compatible with the API of

nanoLambda.

Use Case

Obtain raw sensor data using the user-developed NSP32 interface (USB, Bluetooth or other method) on nanoLambda's SDK 'compatible' platform and obtain spectrum data using nanoLambda's Low-Level API (e.g. Android Platform, Bluetooth connection).

As you already know, you can develop a hardware interface for NSP32 spectral sensor of nanoLambda in a wide variety of ways. As an example, suppose you develop your application using Android phone. NanoLambda's SDK provides a sensor interface API (CrystalPort) via USB, but you might want to develop your own add-on board to access and control the nanoLambda NSP32 spectral sensor directly through Bluetooth functionality and acquire sensor data.

The sensor data acquired through the add-on board (similar in function and purpose to the sensor board in the first case) is converted to spectra using nanoLambda's Low-Level or High-Level API running on the AP of Android Phone. This scenario is possible because nanoLambda provides a Low-Level API built on the Android OS platform already in the ADK.

Anyway, once you have succeeded in developing a hardware interface to nanoLambda's NSP32 spectral sensor, it's an advantage that you can use NSP32 SDK to quickly get spectral data and use it for your application. In this case, however, you should bear in mind that the sensor data of NSP32 spectral sensor acquired through your own interface must have been acquired without any data loss and that the sensor data is valid one. See "**Appendix A: How to validate raw sensor data?**" for detailed information on validation of sensor data directly obtained by the user.

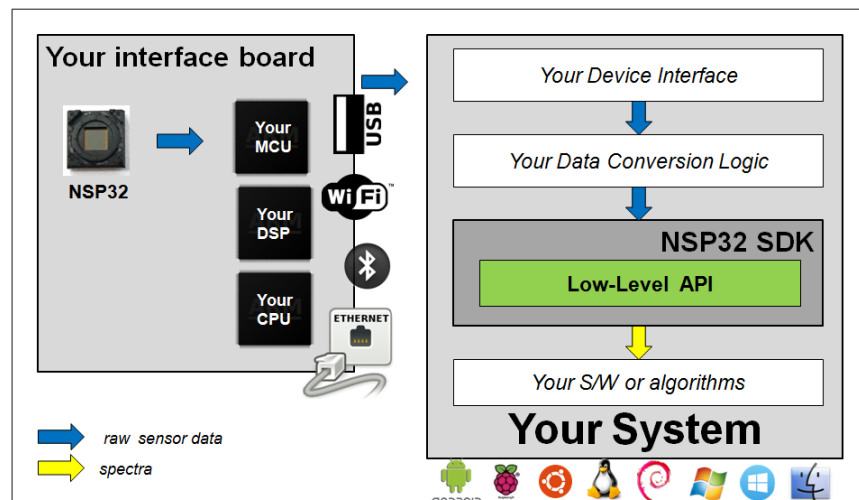


Figure.3 Develop your own interface and calculate spectra with NSP32 APIs.

Chapter 4 Basic Sequence of Operations

Overview

If you have installed the NSP32 SDK on your computer and have decided to use one of the three APIs to develop your application, it is time to understand how the API is orderly because some important API functions have their call sequences internally already defined. For example, the device(sensor) interface object for the NSP32 spectral sensor must be created before you try to acquire sensor data or calculate the spectra with it, and querying the sensor parameters information from the sensor calibration data is possible after the Core Spectrum object is created. You also should be tried to acquire raw sensor data and spectra after setting the optimal shutter speed of the NSP32 spectral sensor to your light conditions.

Regardless of the decisions you make about how to connect the NSP32 spectral sensor to your hardware or system and what level of NSP32 APIs to incorporate in your application S/W, the purpose of nanoLambda's NSP32 ADK is straightforward and clear: *Obtain spectral data of the optical target you are interested in using NSP32 spectral sensors and SDK(APIs) and utilize spectra in your application.* To get the final spectrum correctly, Low-Level or High-Level API functions must be called in the correct order and the valid parameters must be given as inputs to the functions, and the appropriate values should be returned through the output parameters. A detailed description of the input and output parameters for each API function can be found in a separate document: “*NSP32 SDK Reference Manual for Cpp-v1.7.pdf*”.

This chapter will focus on the summarized explanations for you what's recommended sequence of function calls and operations with nanoLambda's NSP32 APIs. If you become familiar with the recommended sequences with Low-Level and High-Level APIs after this chapter, you can start to write your own codes with NSP32 SDK(APIs) for your application.

Note: Don't mix different levels of API

The basic sequence depends on whether you are using Low-Level APIs or High-Level APIs. Do not use Low-Level APIs with High-Level APIs. All sequences are valid even when you write your own application codes with NSP32 ADK. If you are developing your own NSP32 spectral sensor interface board or spectral processing board/system, then you must implement sensor driver (communication protocols, sensor control and data acquisition functions, and other) by yourself. After that, you can use your own sensor interface functions to acquire sensor data, and then you can use nanoLambda's Low-Level API functions to calculate spectra (*your sensor interface driver + nanoLambda's API*).

Note: Open-source (libusb)-based NSP32 USB interface

You should understand that the sensor interface related functions (functions with a prefix 'du-')

like `duConnect()` described in this chapter are implemented with the open source USB library (libusb, <http://libusb.info/>) by nanoLambda and nanoLambda's own communication protocol. Whenever you try to develop your own NSP32 spectral sensor interface by yourself, almost everything on the sensor interface at hardware level is your responsibility. Once you succeed to acquire NSP32 spectral sensor data, then you can use spectra calculation functions in NSP32 SDK (Low-Level API).

Note: Match sensor IDs

Your NSP32 ADK includes a sensor calibration file in addition to the NSP32 spectral sensor mounted on the ADK board or nanoLambda deliver NSP32 spectral sensor with sensor calibration data (file) for that NSP32 spectral sensor. All NSP32 have their own calibration file and using a calibration file tailored to NSP32 spectral sensor is essential to calculate / acquire accurate spectra. Each calibration data file contains sensor ID information to identify the sensor that is the source of the spectral data.

Therefore, you must always make sure that the ID of the physical NSP32 spectral sensor matches with the sensor ID of the calibration file. If you do not do this, especially when you purchase multiple NSP32 spectral sensors from nanoLambda, you have the possibility to specify/ load a calibration file that does not match a particular NSP32 spectral sensor, and the result will be an inaccurate and unreliable spectra. The following is a thumb rule you should remember when you use NSP32 ADK for development:

- 1) Connect NSP32 spectral sensor board included in the NSP32 ADK to your system.
- 2) Obtain the ID of the physical NSP32 spectral sensor using `duGetSensorID()` or your function.
- 3) Use the `csGetSensorID()` function to obtain the sensor ID from your calibration file.
- 4) Verify that the two sensor IDs are matched.

If it does not match, you have to find a correct calibration file for the physical sensor in your hand from your sensor data repository or from nanoLambda. In fact, you can get the ID of the sensor from which the calibration file was created before without getting the sensor ID by calling the `csGetSensorID()` function at software level. This is because all calibration files are created according to the following naming rule by nanoLambda: "*sensor_SENSOR_ID.dat*". You will know in advance from the calibration file name which NSP32 spectral sensor the corresponding calibration file should correspond with.

With High-Level API

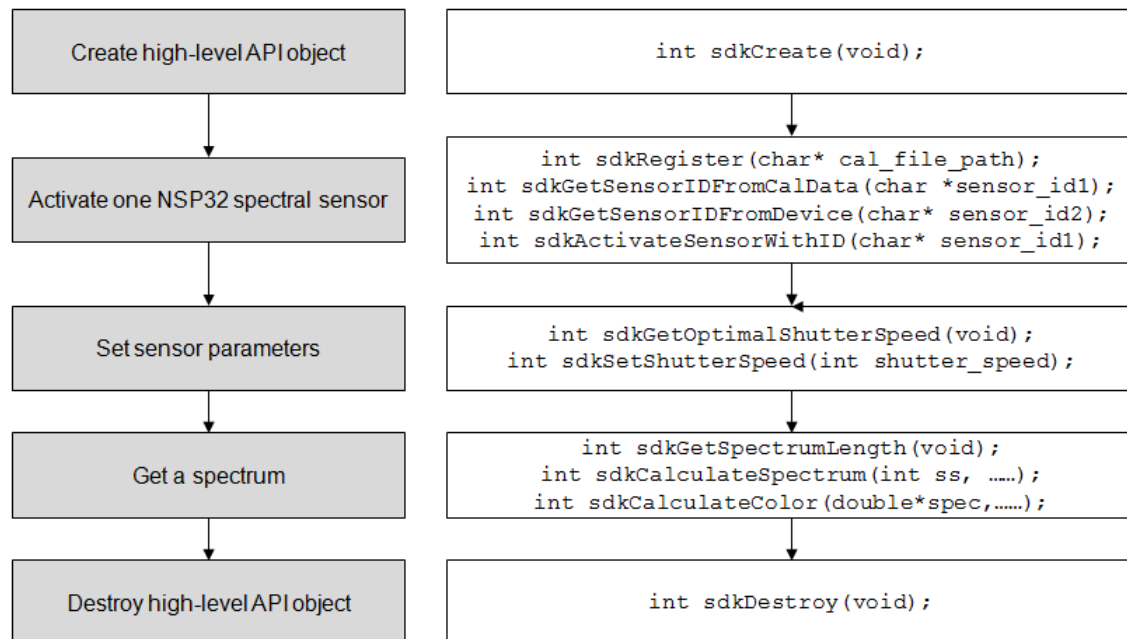
Header Files

```
#include "nsp_base_api.h"
#include "nsp_device_def.h"
#include "nsp_sdk_api.h"
```

Static/Shared Libraries

CrystalBox.lib, CrystalBox.dll

Sequence for spectra calculation (Commentary& Function calls)



Create High-Level API Object

Before attempting to acquire spectra using the NSP32 spectral sensor, you must first create a High-Level API object. The `sdkCreate()` function internally creates the device interfaces that forms the physical connection to your NSP32 spectral sensor (s) (USB connection in the case of the NSP32 ADK) and a core spectrum object that is responsible for spectra computation. You can call `sdkCreate()` multiple times, but if one API object already created, the API object creation function will return with no further creation related to sensor connection and internal object creation. The connection to multiple sensors is managed internally by the High-Level API, so you don't have to worry about that.

C/C++:

```
char* sensor_cal_file_path = "../config/sensor_Y8585-1-85-85-0.dat";
int ret_val = sdkCreate(void);
if( ret_val == NSP_RETURN_VALUE_SUCCESS ){
    int total_sensors = sdkRegister(sensor_cal_file_path);
    // do something more ...
}
```

Activate One NSP32 Spectral Sensor

Your system may have one or more NSP32 spectral sensors connected and you can register one or more sensor calibration data to the sensor data container inside your API using the `sdkRegister()` function. However, if you want to acquire a spectrum from a particular NSP32 spectral sensor at

some point, you must explicitly make that sensor active. You can activate a specific NSP32 spectral sensor using the `sdkActivateSensorWithID()` function, and each time you call `sdkCalculateSpectrum()` to get the spectrum, the API will get the sensor data from the active sensor, calculate one spectrum using the sensor data and the calibration data, and return it to you.

C/C++:

```
char* sensor_id1[SENSOR_ID_STRING_LENGTH];
char* sensor_id2[SENSOR_ID_STRING_LENGTH];

int ret_val = 0;
int total_sensors = sdkGetMaxSensorCount();
int id_size = sdkGetSensorIDFromCalData(str_sensor_id1);
ret_val = sdkActivateSensorWithID(str_sensor_id1);
ret_val = sdkGetSensorIDFromDevice(str_sensor_id2);
if( strcmp(str_sensor_id1, str_sensor_id2, SENSOR_ID_STRING_LENGTH)==0)
    printf("Two sensor IDs are matched. !!\n");
else
    printf("[error] Two sensor IDs are not matched.\n");
```

Set Sensor Parameters

The sensor parameters that you should use with care in your application development are: 1) shutter speed, 2) ADC gain, and 3) ADC range. The ADC gain and ADC range values are set automatically within the High-Level API. So, if you develop your application using a High-Level API, you don't need to worry about because there is no function to do a direct control of these parameters. Shutter speed (exposure time) is a parameter that you use frequently in your application and is a very important parameter that critically determines the quality of the spectrum from your NSP32 spectral sensor. The shutter speed, which guarantees the maximum quality of the spectrum, is dependent on your lighting environment and it is inevitable that the shutter speed value will change when the lighting conditions change. Optimal or near optimal shutter speed values can be found in two ways: 1) you can visually assess the quality of the spectral data after change the shutter speed and repeatedly do this, or 2) find the shutter speed using the function `sdkGetOptimalShutterSpeed()` in API.

C/C++:

```
int optimal_ss = sdkGetOptimalShutterSpeed();
if( optimal_ss == NSP_DEVICE_AE_SATURATION_INDICATOR ) // saturated
    printf("Almost every filters are saturated.\n");
else if( optimal_ss == NSP_DEVICE_MIN_SS ) // almost saturated
    printf("Not saturated but light source is TOO BRIGHT.\n");
else if( optimal_ss == NSP_DEVICE_MAX_SS )
    printf("Light source is too WEAK.\n");
else
    printf("Optimal shutter speed = %d\n", optimal_ss);

// set shutter speed to NSP32 spectral sensor
sdkSetShutterSpeed(optimal_ss);

// if you want to know actual exposure time of optimal_ss
int master_clock = NSP_DEVICE_MASTER_CLOCK;
double exposure_time = 0; // msec
sdkShutterSpeedToExposureTime(master_clock, optimal_ss, &exposure_time);
```

Note: Valid filters vs Monitoring filters

There are two different kinds of filters on the NSP32 spectral sensor: 1) valid filters, 2) monitoring filters. To calculate spectra, NSP32 SDK functions are using valid filters only. Monitoring filters are using for internal/proprietary purposes by nanoLambda, and don't be exposed to you.

Get a Spectrum

Now you are ready to acquire a spectral. A spectral is simply a one-dimensional array of spectral response values of NSP32 spectral sensor, stored as "doubles". The High-Level function to compute the spectrum is `sdkCalculateSpectrum()`.

When you call the `sdkCalculateSpectrum()` function, this function will acquire an averaged NSP32 spectral sensor data to internal buffer, reconstructs a spectra with both sensor and sensor calibration data, and then will return the spectral to your application. Additionally, the duration of your call to `sdkCalculateSpectrum()` could be delayed depends on acquisition parameters like the shutter speed(exposure time) and the frame count for averaging. So, your application may be blocked until the current spectral computation has completed.

C/C++:

```
int shutter_speed = 30;
int frame_averages = 50;
double* spectrum_out = NULL;
double* wavelength_out = NULL;
int spectrum_size = sdkGetSpectrumLength();

// allocate memory for spectrum and wavelength data
spectrum_out = (double *)malloc(spectrum_size*sizeof(double));
wavelength_out = (double *)malloc(spectrum_size*sizeof(double));
if( spectrum_out && wavelength_out ){
    // calculate spectrum
    int ret_val = sdkCalculateSpectrum(shutter_speed, frame_averages,
                                     spectrum_out, wavelength_out);
}
if( spectrum_out ) free((void *)spectrum_out);
if( wavelength_out ) free((void *)wavelength_out);
```

Destroy High-Level API Object

When your application is ready to terminate, call the `sdkDestroy()` function to destroy SDK object in your application space.

C/C++:

```
int ret_value = sdkDestroy();
```

With Low-Level API

Header Files

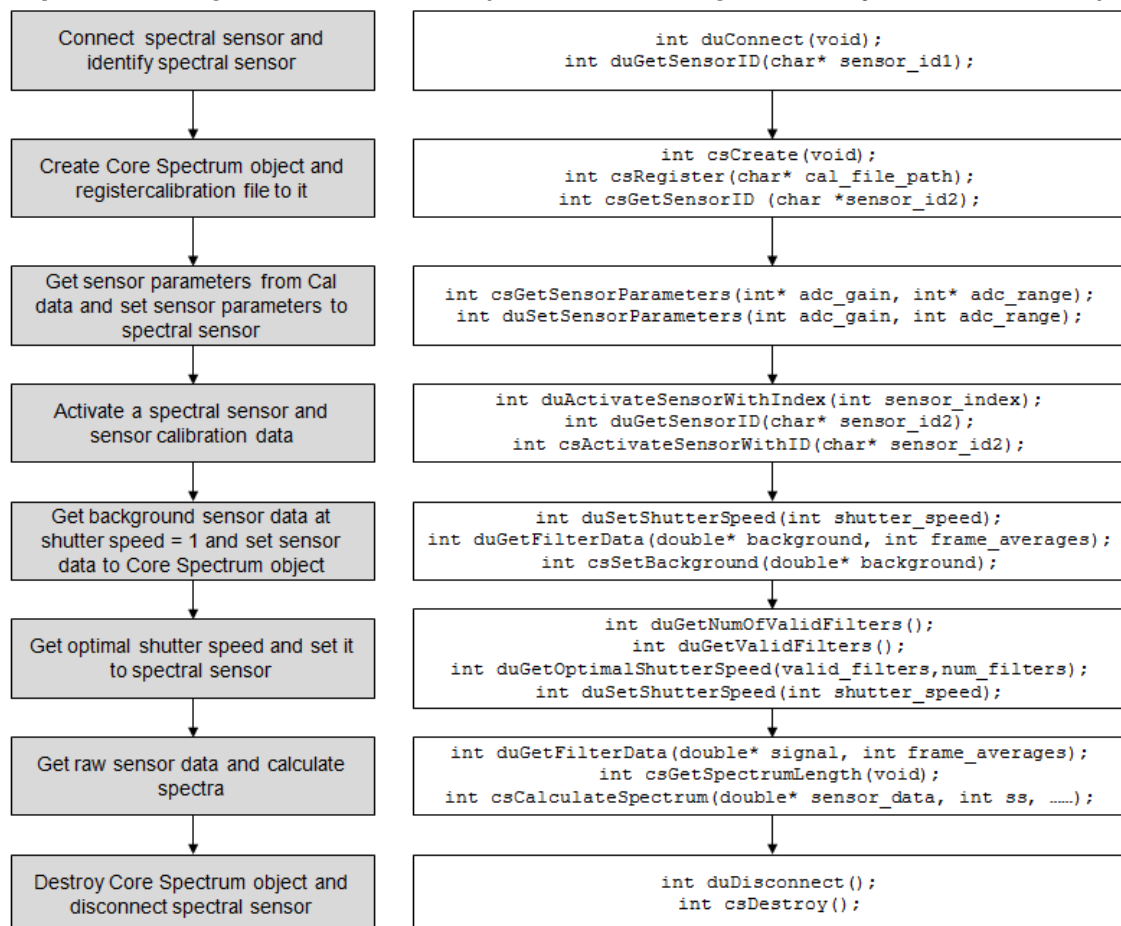
```
#include "nsp_device_def.h"
#include "nsp_base_api.h"
#include "nsp_device_api.h"
#include "nsp_core_api.h"
```

Static/Shared Libraries

CrystalBase.lib, CrystalCore.lib, CrystalPort.lib
 CrystalBase.dll, CrystalCore.dll, CrystalPort.dll

If you have a NSP32 ADK board connected to your system through the USB port and you want to perform a series of tasks to acquire sensor data and spectra calculation using nanoLambda's Low-Level API functions, you can refer to the following sequence to write your own codes.

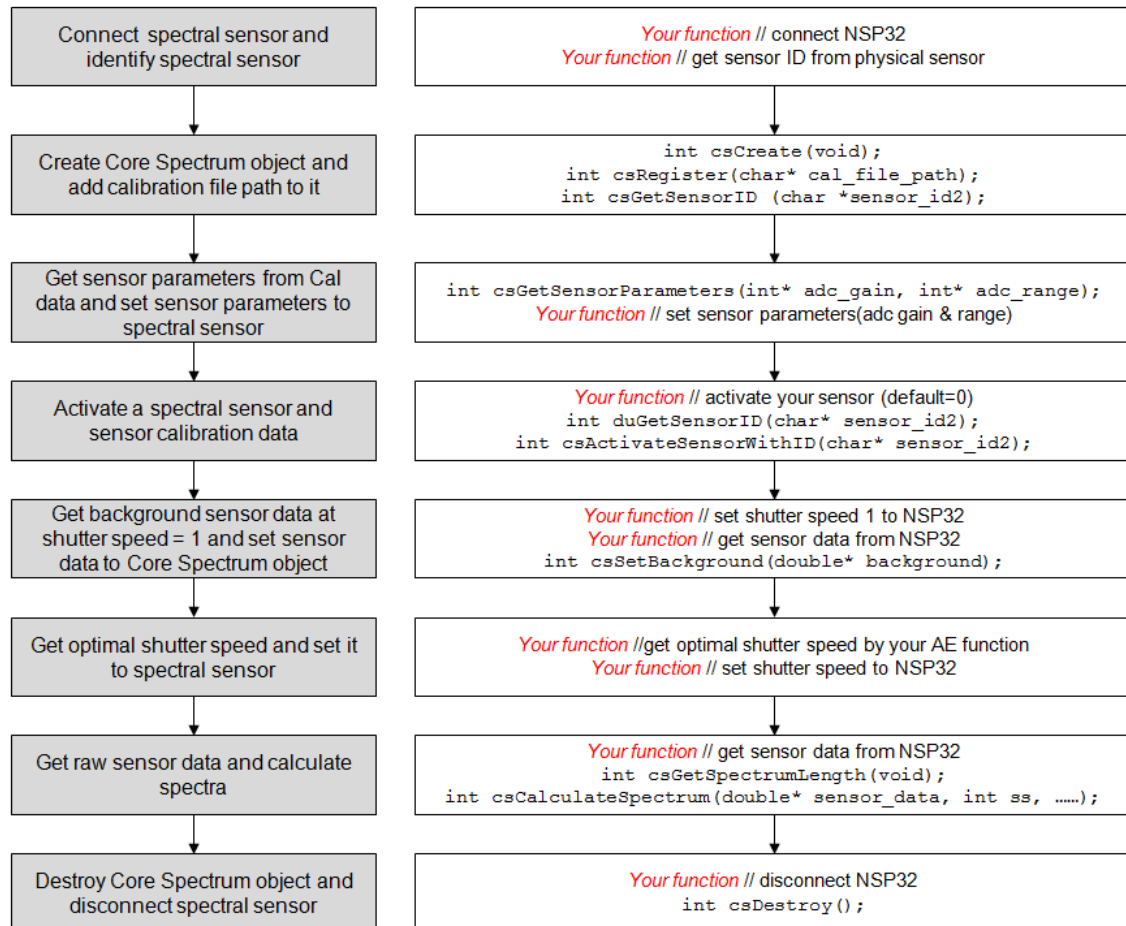
Sequence for spectra calculation (*sensor data acquisition by Low-Level API*)



If you are developing a NSP32 spectral sensor interface for nanoLambda's NSP32 spectral sensor at hardware level, you need to refer to the sequence below. That is, from the spectra calculation point of view, you will acquire NSP32 sensor data through your own sensor interface at hardware level and

you will perform spectra calculation using nanoLambda Low-LevelAPI function. Of course, you must keep in mind that the raw sensor data of the 2048 size char array must be converted to a 1024 size double type array within the your sensor data acquisition function. In the picture below, 'Your function' marked with red color means that you have to implement the function yourself.

Sequence for spectra calculation (*sensor data acquisition by your own interface*)



Connect and Identify NSP32 Spectral Sensor

You can use the Low-LevelAPI function `duConnect()` to internally create a connection to all NSP32 spectral sensors (actually, ADK boards through USB connection) connected to your system. Information on the number of NSP32 spectral sensors connected to your system can be obtained with the `duGetTotalSensors()` function. You can identify the entire NSP32 spectral sensor(s) connected to your system by activating a specific spectral sensor with a sensor index (0 to total number of sensors-1) and then querying the sensor's unique ID with the `duGetSensorID()` function. Although one or more NSP32 spectral sensors can be connected to your system at the same time, spectra data for your application can not be obtained from all NSP32 spectral sensors at the same time. In other words, you can get one spectra from one NSP32 spectral sensor at a time in sequence.

C/C++:

```
char sensor_id_str[SENSOR_ID_STRING_LENGTH];
int optimal_ss = duConnect();
int total_sensors = duGetTotalSensors();
int max_allowed_sensors = duGetMaxSensorCount();
for(int sensor_index=1:sensor_index<total_sensors:sensor_index++)
{
    duActivateSensorWithIndex(sensor_index);
    duGetSensor(sensor_id_str);
    printf("%d-th sensor ID = %s\n", sensor_index, sensor_id_str);
}
```

Create Core Spectrum Object

For spectral calculations, you must create a Core Spectrum object and register a unique calibration file for the NSP32 spectral sensor on that object. If you want to use multiple NSP32 spectral sensors, you must register a calibration file for all sensors in the Core Spectrum object. The order of registration of the calibration file is not critical, and then a specific calibration file can be activated and included in the spectral calculation using the sensor ID.

```
int csCreate(void);
int csRegister(char* cal_file_path);
```

C/C++:

```
int ret_val = csCreate(); // create Core Spectrum object
char* cal_file_path = "../config/sensor_Y8585-1-85-85-0.dat";
ret_val = csRegister(cal_file_path);
```

Get/Set Sensor Parameters

Because the application conditions of the NSP32 spectral sensor vary from user to user, you need to find and configure the sensor parameters that are optimal for your particular application environment (especially, for your light source or signal source). There are three sensor parameters that you can query or set: 1) shutter speed, 2) ADC gain, and 3) ADC range. Querying for ADC gain and ADC range is allowed, but it is prohibited to set to the values you choose because sensor calibration data that is paired with all NSP32 spectral sensors is generated under a specific ADC gain and range setting. If you try to set NSP32 spectral sensor to values other than the ADC gain and range values when creating the sensor calibration data and to acquire the spectra, you will definitely get an inaccurate spectra. So the only sensor parameter you can set for your application is shutter speed.

C/C++:

```
int current_ss = duGetShutterSpeed(); // query current shutter speed
int num_valid_filters = duGetNumOfValidFilters();
int* valid_filters = duGetValidFilters();
int
optimal_ss=duGetOptimalShutterSpeed(valid_filters,num_valid_filters);
int ret_val = duSetShutterSpeed(optimal_ss);
int adc_gain = 0, adc_range = 0;
// get sensor parameters from calibration data
ret_val = csGetSensorParameters(&adc_gain, &adc_range);
```

```
// set sensor parameters to NSP32
Ret_val = duSetSensorParameters(adc_gain, adc_range);
```

Activate Sensor and Cal Data

Activating the sensor determines the physical sensor to acquire sensor data and the calibration data corresponding to that sensor. If you are using more than one sensor for the evaluation of the NSP32 ADK or for your own application, the corresponding sensor and its corresponding calibration data must be active to get accurate spectra from different NSP32 spectral sensors. Sensor parameters query or setting by Low-Level API function calls, sensor data acquisition, and spectral computation are all performed in conjunction with the currently activated NSP32 spectral sensor and its calibration data. If you use the Low-Level API, you must explicitly activate both NSP32 spectral sensor (with integer index or string ID) and the calibration data.

C/C++:

```
int ret_value = 0;
int sensor_index = 0; // 0-based.
char* sensor_id_str = "Y8585-1-85-85-0";
char sensor_id_from_sensor[SENSOR_ID_STRING_LENGTH];
char sensor_id_from_cal_data[SENSOR_ID_STRING_LENGTH];
ret_value = duActivateSensorWithIndex(sensor_index);
if( ret_value == NSP_RETURN_VALUE_FAILURE )
    return; // fail to activate one sensor with index
OR
ret_value = duActivateSensorWithID(sensor_id_str);
if( ret_value == NSP_RETURN_VALUE_FAILURE )
    return; // fail to active one sensor with sensor ID
duGetSensorID(sensor_id_from_sensor);
csActivateSensorWithID(sensor_id_from_sensor); // activate cal.Data
csGetSensorID(sensor_id_from_cal_data);
if( strcmp(sensor_id_from_sensor, sensor_id_from_cal_data) != 0 )
    return; // error, two sensor IDs are not matched !
```

Get/Set Background Data

The purpose of the background data is to remove the inherent dark noise of the NSP32 spectral sensor before performing spectrum reconstruction using sensor data obtained from the NSP32 spectral sensor. The correction of the dark noise is to ensure the linear response of the NSP32 spectral sensor and minimizes the side peaks, resulting in an accurate and reliable spectrum. The background data should always be obtained under the condition of **shutter speed = 1** and the frame count for averaging should be a sufficiently large number (nanoLambda recommends using at least 30 frames) to minimize the effect of the variations from NSP32 spectral sensor and ambient.

C/C++:

```
double raw_sensor_data[1024];
int frame_averages = 50;
int ret_val = duSetShutterSpeed(1);
ret_val = duGetFilterData(raw_sensor_data, frame_averages);
ret_val = csSetBackground(raw_sensor_data);
```


Get/Set Optimal Shutter Speed

The auto exposure function is similar to the auto exposure function of general consumer cameras. The auto-exposure feature can be used to automatically determine the optimal exposure to obtain the best spectrum without any user intervention or decision before acquiring one spectral data. You can use the `duGetOptimalShutterSpeed ()` function to get the shutter speed value when the output of the filter with the maximum response under any light source approaches NSP32 spectral sensor maximum dynamic range value (4096). To use `duGetOptimalShutterSpeed ()` function, you have to put the information about valid filters of your NSP32 spectral sensor. If an argument `valid_filters` is NULL, `duGetOptimalShutterSpeed ()` function will use default valid filters to have an optimal shutter speed. The calculation of the optimal shutter speed by the AE function depends on the intensity of the light source in the environment in which the NSP32 spectral sensor is present.

C/C++:

```
int num_valid_filters = duGetNumOfValidFilters();
int* valid_filters = duGetValidFilters();
int
optimal_ss=duGetOptimalShutterSpeed(valid_filters,num_valid_filters);
int ret_val = duSetShutterSpeed(optimal_ss);
```

Get Raw Sensor Data and Calculate Spectra

Unlike the High-Level API, which acquires spectra with a single call to a function (`sdkCalculateSpectrum ()`), when you use the Low-Level API for your application, the spectra you want can be obtained in two steps. That is, in the first step, you obtain sensor data from the NSP32 spectral sensor by calling the `duGetFilterData ()` function. In the second step, you perform spectra computation through `csCalculateSpectrum ()`. In order to optimally sense the signal (spectrum) you are interested in, you must adjust the shutter speed (exposure time) of the NSP32 spectral sensor to your signal source.

If the intensity of the signal source being sensed is weak, it is necessary to increase the shutter speed to increase the exposure time, or to decrease the shutter speed by reducing the exposure time to eliminate the saturation possibility if the intensity of the signal source is sufficiently high. For your reference, when an appropriate level of signal strength is guaranteed, you can use the Low-Level function(`duGetOptimalShutterSpeed ()`) to find the optimal shutter speed more easily.

C/C++:

```
int num_valid_filters = duGetNumOfValidFilters();
int* valid_filters = duGetValidFilters();
int
optimal_ss=duGetOptimalShutterSpeed(valid_filters,num_valid_filters);
int frame_averages = 50;
double sensor_data_ss_1[1024], sensor_data_ss_N[1024];

// acquire background data at shutter speed = 1
int ret_val = duSetShutterSpeed(1);
ret_val = duGetFilterData(sensor_data_ss_1, frame_averages);

// set background data to Core Spectrum object
ret_val = csSetBackground(sensor_data_ss_1);
```

```
// acquire target sensor data at shutter speed = N
ret_val = duSetShutterSpeed(optimal_ss);
ret_val = duGetFilterData(sensor_data_ss_N, frame_averages);

// get spectrum length and calculate spectrum
int spec_length = csGetSpectrumLength();
double* spec_data = (double *)malloc(sizeof(double)*spec_length);
double* wave_data = (double *)malloc(sizeof(double)*spec_length);
if( spec_data ){
    ret_val = csCalculateSpectrum(sensor_data_ss_N, optimal_ss,
                                spec_data, wavelength_data);
}
.....
```

Destroy Core Spectrum Object

When your application is ready to terminate, call the `csDestroy()` function to destroy your Core Spectrum object in your application space.

C/C++:

```
int ret_val = csDestroy();
```

Disconnect NSP32 Spectral Sensor

When your application is ready to terminate, call the `duDisconnect()` function to disconnect your NSP32 spectral sensor from your application space.

C/C++:

```
int ret_val = duDisconnect();
```

Chapter 5 Acquisition Parameters

Overview

This chapter presents a complete list of all available acquisition parameters. An acquisition parameter is a setting that controls some aspect of the conditions under which a spectrum is acquired.

Note: NSP32 SDK/ADK support USB connection only

All of the functions used here are based on the assumption that you use the SensorInterface object included in nanoLambda's NSP32 SDK. The NSP32 SDK includes a USB-based connection / protocol for control / data acquisition of NSP32 spectral sensor on the ADK board. All of the NSP32 SDK functions work through these USB-based connections / protocols. **If you develop / implement your own NSP32 sensor interface at the hardware level, each parameter must be set / queried by your functions. And, it is up to you to validate NSP32 spectral sensor data acquired by your sensor interface.**

Shutter Speed

Shutter speed is the exposure time actually and controls the amount of incident light to the sensor. In other words, you can control the exposure time of the sensor's light source by adjusting the shutter speed of the NSP32 spectral sensor. Increasing the shutter speed value increases the exposure time, which helps detect weak signals, but at the same time increases the effect of ambient light other than the light you are interested in. In addition, the dark current value of NSP32 spectral sensor itself also increases, so you can get inaccurate spectra.

Therefore, it is not desirable to increase the shutter speed value indefinitely. By increasing the intensity of the light source or narrowing the distance between NSP32 spectral sensor and the light source of you before increasing the shutter speed value to get a sufficient amount of light, NSP32 spectral sensor generates the right spectrum. After optimizing the optical settings like optimal light source intensity and optical path geometry as much as possible, it is a reasonable approach to obtain the spectrum by adjusting (increasing) the shutter speed value if necessary.

The shutter speed can be set using the `duSetShutterSpeed ()` and `sdkSetSutterSpeed ()` functions. The shutter speed currently set on the NSP32 spectral sensor can be obtained using `duGetShutterSpeed ()` and `sdkGetShutterSpeed ()`. The shutter speed is closely related to the optical settings associated with your application, which must be increased or decreased depending on the intensity of the optical target (e.g., light source) you are using or interested in.

- If the intensity of the signal of optical target is strong, setting the shutter speed to a high value can easily saturate each pixel of NSP32 spectral sensor.
- If the intensity of the signal of optical target is too weak, setting the shutter speed value too low increases the likelihood of obtaining an inaccurate spectrum (almost noise).

The optimal shutter speed is the shutter speed at which the response (intensity) of the nano-filter having the highest sensitivity to the corresponding light among the valid filters excluding the monitoring filters is maximized. Shutter speed is an integer value with no units, but you can use

`duShutterSpeedToExposureTime()` function to convert the shutter speed value to exposure time. You can also get 'approximated' shutter speed value by converting the exposure time to shutter speed (`duExposureTimeToShutterSpeed()`).

Note: Shutter speed to Exposure time

You can convert the shutter speed to exposure time or the exposure time to shutter speed using the two formulas given below by yourself. In these equations, MCLK is the master clock from the MCU to NSP32 spectral sensor. If you have developed a hardware interface for the NSP32 spectral sensor, you must set your correct MCLK to the value you specify to get the correct conversion value. For reference, the MCLK at nanoLambda's NSP32 ADK is 5MHz.

```
MCLK = 5; // 5MHz
exposure_time = (((384+16*(32+1))*shutter_speed)+160)/(MCLK*1000);
shutter_speed = (((exposure_time_val*(MCLK*1000)-160)-384)/(16*(32+1)));
```

Note: Frame rate depends on Shutter speed

Using a high shutter speed will lower the frame rate of the NSP32 spectral sensor and a lower shutter speed will increase the frame rate. The NSP32's MCLK is up to 32 MHz. The controllable MCLK will depend on the main clock of your processor (MCU, AP, or other).

Function call:

// Low-Level API

```
int duSetShutterSpeed(int shutter_speed);
int duGetShutterSpeed(void);
```

// High-Level API

```
int sdkSetShutterSpeed(int shutter_speed);
int sdkGetShutterSpeed(void);
```

Auto-Exposure

The auto exposure function is similar to the auto exposure function of general consumer cameras. The auto-exposure feature can be used to automatically determine the optimal shutter speed to obtain the best spectrum without any user intervention or manual decision before acquiring one spectral data. You can use `duGetOptimalShutterSpeed()` or `sdkGetOptimalShutterSpeed()` functions to get the shutter speed value when the output of the filter with the maximum response under any spectral sensing conditions is approach to NSP32 spectral sensor maximum dynamic range value (4096). The calculation of the optimal shutter speed by the AE function depends on the intensity of the light source in the environment in which the NSP32 spectral sensor is present. If the intensity of the light source is high, the optimal shutter speed can be calculated quickly, and if the intensity of the light source is weak, it takes a lot of time (e.g., tens of seconds) to calculate the optimal shutter speed.

Note: Find optimal shutter speed with AE

The details related to the optimal shutter speed calculation function based on Auto-Exposure are all implemented internally in NSP32 SDK and hidden from you, and you only allowed to obtain the optimal shutter speed value by calling the related function `duGetOptimalShutterSpeed()` or `sdkGetOptimalShutterSpeed()`. To use `duGetOptimalShutterSpeed()` or `sdkGetOptimalShutterSpeed()` functions, you have to put the information about valid filters of your NSP32 spectral sensor. If an argument `valid_filters` is NULL, `duGetOptimalShutterSpeed()` or `sdkGetOptimalShutterSpeed()` functions will use default valid filters to have an optimal shutter speed.

Of course, you can use the trial-and-error approach to find the optimal shutter speed value for your application.

Function call:

```
// Low-Level API
int duGetOptimalShutterSpeed(int* valid_data,int num_valid_filters);

// High-Level API
int sdkGetOptimalShutterSpeed();
```

ADC Settings(ADC gain & range)

The ADC in NSP32 spectral sensor serves to output the filter response in digital form. The ADC on the NSP32 spectral sensor provides 10-bit or 12-bit settings (ADC resolution or precision). The NSP32 ADC version 1.7 uses a fixed 12-bit ADC resolution and does not allow you to change it.

And, NSP32 spectral sensor allows the ADC to set two different conversion gains: gain 1x (value = 1) and gain 4x (value = 0). Conversion gain is simply the effect of a single charge sensed by the sensor on the change in output voltage. In other words, the NSP32 spectral sensor set to gain 4x responds theoretically four times more sensitive to changes in external signal than gain 1x. The NSP32 ADK version 1.7 is set to gain 1x and will support gain 4x in later versions. ADC gain also not allowed change by you.

ADC range is the index value for the LUT that determines the operating voltage range of the ADC.

Repeatedly speaking, NSP32 spectral sensor supplied to you is paired with the supplied sensor calibration data. Here is one thing you should be aware of in your application. You can obtain accurate and reliable spectra by setting your physical NSP32 spectral sensor with the sensor parameters in the sensor calibration file for that NSP32 spectral sensor. So, you can use API functions to change the sensor parameters (ADC settings) to familiarize yourself with the NSP32 spectral sensor, but when you actually want to get spectra in your application, you must make sure that your sensor is set to the sensor parameters (ADC settings) obtained from the sensor calibration data correctly.

Function call:

```
// Low-Level API
int csGetSensorParameters(int* adc_gain,int* adc_range);//from cal. data
```

```
int duSetSensorParameters(int adc_gain,int adc_range);//to physical
sensor
int duGetSensorParameters(int* adc_gain,int* adc_range);//from physical
sensor
```

Frame Count for Averaging

Spectral reconstruction should be performed after acquiring and averaging multiple sensor data from NSP32 spectral sensor to improve the S / N of the spectrum or to minimize the effect of flickering of the illumination you are using. nanoLambda recommends that you average at least 30 frames of sensor data. Averaging is performed on each filter unit (pixel) for all 1024 filters that make up one frame. For example, assuming that 50 averaging are performed, the average value of the value of each n-th filter is stored in the n-th position of the last frame.

Note: Frame averaging

There is no independent function to set the number of frames for averaging. You can set the frame count as an input argument to `duGetFilterData()`, which is a Low-Level function used to obtain raw filter data, or `sdkCalculateSpectrum()`, which is a High-Level function used to calculate the spectrum.

Function call:

//Low-Level API

```
int duGetFilterData(double* filter_data, int frame_averages);
```

// High-Level API

```
int sdkCalculateSpectrum(int shutter_speed, int frame_averages, double*
spec_out, double* wavelength_out);
```

Chapter 6 High-Level API

Overview

This chapter describes the typical sequence of operations that the application must perform to control nanoLambda's NSP32 spectral sensor and acquire spectra from it. The High-Level functions described here are most abstracted ones to reduce the development burdens of user as many as possible. To build your application, you must include one header file ('nsp_sdk_api.h') and link to one static library ('CrystalBox.lib'). This chapter provides exact syntax for each function call with C/C++ code snippet, including all parameters and data types.

Note: NSP32 SDK/ADK supports USB connection only

All of the functions used here are based on the assumption that you use NSP32 ADK. The NSP32 SDK includes a USB-based connection / protocol for control / data acquisition of the NSP32 spectral sensor on the ADK board. All of the NSP32 SDK functions work through these USB-based connections / protocols. If you develop / implement your own NSP32 sensor interface at the hardware level, there is no way to use High-Level API for you.

Create Sensor Interface

Before you control your NSP32 spectral sensor, you must create one High-Level SDK object. The details are hidden from you at this API level, but during creation of SDK object, USB connection to sensor(s) is created and sensor registers are set with default parameters by `duConnect()` Low-Level function which attempts to enumerate the connected NSP32 spectral sensors at every available USB address and create USB connection(s) to them. If you receive a value (> 0), you can conclude that one or more NSP32 spectral sensor is present in the system. And next, SDK object is created by `csCreate()` function and will return `NSP_RETURN_VALUE_SUCCESS` if it succeeded or `NSP_RETURN_VALUE_FAILURE` if it failed.

Note: Singleton SDK object

Your application must create only one instance of SDK object. If you try to create another instance of SDK object, `sdCreate()` will return `NSP_RETURN_VALUE_SUCCESS` value **without** additional creation of this object. All interactions with every nanoLambda spectral sensor(s) which are attached to your computer must be performed within this one single executable/application.

Function call:

```
int sdkCreate(void);
```

C/C++:

```
int ret_val = sdkCreate();
if( ret_value == NSP_RETURN_VALUE_SUCCESS){
    // control spectral sensor
}
```

Add a Calibration Data to Data Repository

Once High-LevelSDK object is created successfully, next, you must registersensor calibration file of one sensor to internal repository of SDK object by providing a sensor calibration data file path as an argument to `sdRegister()` function. If you have a return value larger than 0, then this addition is succeeded and from now on, you can use this sensor and calibration data for your application by activating it with `sdActivateWithSensorIndex()` or `sdActivateWithSensorID()` function.

The `sdRegister()` function normally returns an integer from 0 – N, indicating the number of NSP32 spectral sensors in the sensor data repository which managed by SDK internally. If an error occurred during the registration of calibration file to the repository, this method will returns “0” or negative value as exception code.

Function call:

```
int sdRegister(std::string sensor_cal_data_path);
```

C/C++:

```
std::string sensor_cal_path = "./config/sensor_Y8585-1-85-85-0.dat";
int total_count_of_active_sensors = sdRegister(sensor_cal_path);
if( total_count_of_active_sensors > 0){
    // control spectral sensor
}
```

Activate a NSP32 spectral sensor

To use a spectral sensor, user must activate a physical NSP32 spectral sensor of interest and the matched sensor calibration data first by calling `sdActivateSensorWithIndex()` or `sdActivateSensorWithID()` function with sensor index or sensor ID as an input argument. If there are multiple NSP32 spectral sensors in your system, this is only way to switch between different NSP32 spectral sensors, to prepare a NSP32 spectral sensor with a unique calibration data and to calculate spectra.

The `sdActivateSensorWithIndex()` or `sdActivateSensorWithID()` method normally returns an integer `NSP_RETURN_VALUE_SUCCESS` or `NSP_RETURN_VALUE_FAILURE` indicating success or failure, respectively.

Function call:

```
int sdActivateSensorWithIndex (int sensor_index);
int sdActivateSensorWithID (const char* sensor_id_str);
```

C/C++:

```
int sensor_index = 0; // 1'st sensor
const char* sensor_id_str = "Y8585-1-85-85-0";
int ret_val = 0;
ret_value = sdActivateSensorWithIndex(sensor_index); //activate with
index
ret_value = sdActivateSensorWithID(sensor_id_str); //activate with ID
```


Get Activated Sensor Information

If you wish to identify which NSP32 spectral sensor is currently activated and you are communicating with, there are two High-Level functions that help identify NSP32 spectral sensor by returning sensor's unique ID. You can get a sensor ID from physical NSP32 spectral sensor or you can get a sensor ID from sensor calibration data. If you activated a NSP32 spectral sensor via `sdkActivateSensorWithIndex()` or `sdkActivateSensorWithID()`, then two sensor IDs will be exactly matched. Usually, two sensor IDs have to be matched. If not, the correctness of calculated spectra will not be guaranteed:

Function call:

```
int sdkGetSensorIDFromDevice (const char* sensor_id_str);
int sdkGetSensorIDFromCalData (const char* sensor_id_str);
```

C/C++:

```
char sensor_id1[SENSOR_ID_STRING_LENGTH];
char sensor_id2[SENSOR_ID_STRING_LENGTH];
int ret_value = 0;
ret_value= sdkGetSensorIDFromDevice(sensor_id1);
ret_value = sdkGetSensorIDFromCalData(sensor_id2);
```

Get/Set Sensor Parameters

Normally you will want to set one or more acquisition parameters controlling the conditions under which spectra are obtained. “*Chapter 5: Acquisition Parameters*” lists all available parameters. Once again, three parameters (shutter speed, ADC gain value and ADC range value) are essential parameters for sensor (nano-filter response) data and spectra acquisition. Except shutter speed, both ADC gain and ADC range values are determined by sensor calibration data and set up internally, and you don't need to care about ADC gain and range values at High-Level API. If you want to control ADC gain and range values directly, you must use Low-Level API. Here we show how to set an acquisition parameter (shutter speed) with High-Level function.

Function call:

```
int sdkGetShutterSpeed (void);
int sdkSetShutterSpeed(int shutter_speed);
int sdkGetShutterSpeedLimits(int* min_val, int* max_val);
```

C/C++:

```
// query current shutter speed
int old_shutter_speed = 0;
old_shutter_speed = sdkGetShutterSpeed();

// set shutter speed with new value
int new_shutter_speed = 100;
int result_val = sdkSetShutterSpeed(new_shutter_speed);

// you can check the available range of shutter speed
int min_shutter_speed = 0;
int max_shutter_speed = 0;
sdkGetShutterSpeedLimits(&min_shutter_speed, &max_shutter_speed);
```

Find Optimal Shutter Speed

Actually, shutter speed is exposure time, that is, the length of time when NSP32 spectral sensor is exposed to input light. The amount of input light that reaches to NSP32 spectral sensor is proportional to the shutter speed or exposure time. To have correct spectra with NSP32 spectral sensor, you must set shutter speed value to have enough amount of light. The value of shutter speed need to be increased or decreased depends on the power of input light of interest. The ADC resolution of nanoLambda's NSP32 spectral sensor is 12-bit, so maximum DN what you can achieve by adjusting shutter speed is 4096. As thumb of rule, higher DN is better spectra.

We calls a certain shutter speed as 'optimal' shutter speed with which the closest value to maximum DN(4096) is achieved. Due to the different illumination conditions of your spectral sensing experiments with NSP32 spectral sensor, you may don't know what shutter speed is optimal to acquire best spectra. We provides one convenient function that works based on AE(Auto-Exposure) function: `sdkGetOptimalShutterSpeed()`. If you are work in the low light condition, it will take significant time (several tenth seconds) to get 'optimal' shutter speed. If you are work with the too strong light source, NSP32 spectral sensor can be saturated very easily even with minimum shutter speed (1).

`sdkGetOptimalShutterSpeed()` function normally returns an integer from `1~NSP_DEVICE_MAX_SS`, indicating the allowable range of shutter speed. If a found shutter speed value is larger than `NSP_DEVICE_MAX_SS`, `NSP_DEVICE_MAX_SS` is returned and this condition means that light quantity on NSP32 spectral sensor is too low. If too many nano-filters are saturated, then this function returns "-1".

Function call:

```
int sdkGetOptimalShutterSpeed();
```

C/C++:

```
int optimal_shutter_speed = sdkGetOptimalShutterSpeed();
if( optimal_shutter_speed == -1 ) // filter values are saturated
    printf("Illumination is too strong (saturation is happened.)\n");
else if( optimal_shutter_speed == NSP_DEVICE_MAX_SS ) // 1200
    printf("Illumination is too weak.\n");
else
    printf("Illumination is in normal state.\n");
```

Note: Setup illumination for spectral sensing

Illumination can be subject to three conditions:

- it is too dark
- normal
- it is too bright (strong).

You are not guaranteed to get the correct spectrum in 1) or 3). If your lighting condition is 1), you must increase the amount of light reaching the sensor by reducing the distance between NSP32 spectral sensor and the light source, or by increasing the shutter speed value. In case 3), contrary to 1), the amount of light reaching the sensor must be reduced by increasing the distance between NSP32 spectral sensor and the light source or by decreasing the shutter speed value. The requirement

for NSP32 spectral sensor to reliably generate the correct spectra is that none of the 1024 nano-filters will saturate under some sort of light source while the response of each nano-filter should approach the maximum (4096). It is very important for users to find the optimum distance between the light source and NSP32 spectral sensor and the optimum shutter speed value. After finding this optimal condition, you can find the fine tuned conditions (distance between the light source and the spectrum sensor, and shutter speed) that can get the best spectrum by slightly increasing or decreasing the value of the shutter speed. This optimal state ensures accurate spectral acquisition.

Shutter Speed To Exposure time

The nanoLambda NSP32 spectral sensor basically adjusts the exposure time of the sensor through the shutter speed value. But often you may want to use a time unit exposure time instead of a shutter speed value as an integer value without a unit. In this case, the NSP32 ADK provides two conversion functions between shutter speed and exposure time: `sdkShutterSpeedToExposureTime()` and `sdkExposureTimeToShutterSpeed()`. The unit of exposure time is msec.

C/C++:

```
int ret_value = 0;
int MCLK = 5;      // 5MHz, master clock of MCU to sensor
int shutter_speed = 100;
double exposure_time = 0;
ret_value = sdkShutterSpeedToExposureTime(MCLK, shutter_speed,
&exposure_time);

exposure_time = 100;    // 100 msec
shutter_speed = 0;
ret_value = sdkExposureTimeToShutterSpeed(MCLK, exposure_time,
&shutter_speed);
sdkSetShutterSpeed(shutter_speed);
```

Acquire a Spectral

Now you are ready to acquire a spectral. A spectral is simply a one-dimensional array of calibrated sensor response values of NSP32 spectral sensor, stored as “doubles”. The High-Level function to compute the spectrum is `sdkCalculateSpectrum()`.

When you call the `sdkCalculateSpectrum()` function, this function will acquire an averaged sensor data to internal buffer, reconstructs a spectra with both sensor data and sensor calibration data, and then will return the spectral to your application. Additionally, the duration of your call to `sdkCalculateSpectrum()` could be delayed depends on acquisition parameters like the shutter speed(exposure time) and the frame count for averaging. So, your application may be blocked until the current spectral acquisition has completed.

Function call:

```
int sdkCalculateSpectrum(int shutter_speed, int frame_averages,
double* spectrum_out, double* wavelength_out);
```

C/C++:

```
#include "nsp_base_api.h"
#include "nsp_device_def.h"
#include "nsp_sdk_api.h"

int shutter_speed = 30;
int frame_averages = 50;
double* spectrum_out = NULL;
double* wavelength_out = NULL;
int spectrum_size = sdkGetSpectrumLength();
// allocate memory for spectrum and wavelength data
spectrum_out = (double *)malloc(spectrum_size*sizeof(double));
wavelength_out = (double *)malloc(spectrum_size*sizeof(double));
if( spectrum_out && wavelength_out ){
    // calculate spectrum
    int ret_val = sdkCalculateSpectrum(shutter_speed, frame_averages,
                                     spectrum_out, wavelength_out);
}
if( spectrum_out ) free((void *)spectrum_out);
if( wavelength_out ) free((void *)wavelength_out);
```

Destroy Sensor Interface

When your application is ready to terminate, call the `sdkDestroy()` function to destroy SDK object in your application space.

Function call:

```
int sdkDestroy(void);
```

Chapter 7 Low-Level API

Overview

This chapter describes the typical sequence of operations that the application must perform to control nanoLambda's NSP32 spectral sensor and acquire spectra. Because the Low-Level functions described here are less abstracted ones, you can access to and control NSP32 spectral sensor using 'sensor interface object' and also you can calculate spectra using 'core spectrum API'. There are three low level APIs: 1) Base API, 2) Device API, and 3) Core Spectrum API. To use these APIs in your application, you must include header files of these APIs ('nsp_base_api.h', 'nsp_device_api.h', and 'nsp_spectrum_api.h') and link with static libraries (CrystalBase.lib, CrystalCore.lib, and CrystalPort.lib). This chapter provides exact syntax for each function call with C/C++ code snippet, including all parameters and data types.

Note: Prefix for APIs

You can identify functions from these other APIs with different prefixes. For example, base API functions start with 'bs' (base system), sensor (device) interface functions with 'du' (device usb), and Core Spectrum API functions with 'cs' (core spectrum).

Note: NSP32 SDK supports USB connection only

The NSP32 SDK includes a USB-based connection / communication protocols for control / data acquisition of the NSP32 spectral sensor on the ADK board. All of the NSP32 SDK functions work through these USB-based connections / protocols. **If you develop / implement your own NSP32 sensor interface at the hardware level, you don't need to use sensor (device) interface functions in Low-Level API.**

Connect NSP32 Spectral Sensor

Before you start control your NSP32 spectral sensor, you must connect one or more NSP32 spectral sensors to your system via USB connection when you use NSP32 ADK board for your application development. Currently, NSP32 ADK supports USB connection only.

Function call:

```
int duConnect(void);
```

C/C++:

```
int ret_val = duConnect();
if( ret_value == NSP_RETURN_VALUE_SUCCESS){
    // control spectral sensor
}
```

Activate Physical Sensor

To use a NSP32 spectral sensor, user must activate a physical NSP32 spectral sensor of interest

and the matched sensor calibration data object first by calling `duActivateSensorWithID` () function with sensor ID as an input argument. If there are multiple NSP32 spectral sensors in your system, this is only way to switch between different NSP32 spectral sensors, to prepare a NSP32 spectral sensor with a unique calibration data and to calculate spectra.

These two functions normally returns an integer `NSP_RETURN_VALUE_SUCCESS` or `NSP_RETURN_VALUE_FAILURE` indicating success or failure, respectively.

Function call:

```
int duActivateSensorWithID (const char* sensor_id_str);
```

C/C++:

```
int sensor_index = 0; // 1'st sensor
const char* sensor_id_str = "Y8585-1-85-85-0";
int ret_val = 0;
ret_value = duActivateSensorWithID(sensor_id_str); //activate with ID
```

Get Total Sensors in System

You can use the `duGetTotalSensors` () function to get information about the total number of NSP32 spectral sensors connected to your system.

Function call:

```
int duGetTotalSensors(void);
```

C/C++:

```
int total_num_of_sensors = duGetTotalSensors();
if( total_num_of_sensors > 0){
}
```

Get Maximum Count of Sensors

You can use the `duGetMaxSensorCount` () function to get about the maximum number of NSP32 spectral sensors allowed to be connected to your system.

Function call:

```
int duGetMaxSensorCount(void);
```

C/C++:

```
int max_count_of_sensors = duGetMaxSensorCount();
if(max_count_of_sensors> 0 ){
}
```

Get Sensor List in System

You can use the `duGetSensorList` () function to get physical sensor ID list in your system. This function will allocate the necessary memory to store whole sensor IDs, put all sensor IDs into string array, and return that string array.

Function call:

```
char**duGetSensorList();
```

C/C++:

```
char** sensor_list_in_your_system;
sensor_list_in_your_system =duGetSensorList();
```

Get Physical Sensor ID

If you wish to identify which NSP32 spectral sensor is currently activated and you are communicating with, there is a Low-Level function that helps identify NSP32 spectral sensor by returning sensor's unique ID. You can get a sensor ID from physical NSP32 spectral sensor with `duGetSensorID()` function. You can also obtain the sensor ID from the sensor calibration file supplied with NSP32 spectral sensor by nanoLambda. You must always keep in mind that the physical sensor ID and the ID from the calibration file must be the same to ensure accurate and valid spectrum calculations.

Function call:

```
int duGetSensorID(const char* sensor_id_str);
```

C/C++:

```
char sensor_id[SENSOR_ID_STRING_LENGTH];
int ret_value = 0;
ret_value = duGetSensorID(sensor_id);
```

Get a List of Connected Sensor IDs

You can use the `duGetSensorList()` function to get whole connected sensor IDs to your system as an array of character array pointer. This function will allocate the necessary memory to store whole sensor IDs (string).

Function call:

```
char**duGetSensorList ();
```

C/C++:

```
char** sensor_ID_list = NULL;
sensor_ID_list = duGetSensorList();
```

Shutter Speed To Exposure Time/ Exposure Time To Shutter Speed

The nanoLambda NSP32 spectral sensor basically adjusts the exposure time of the sensor through the shutter speed value. But often you may want to use a time unit exposure time instead of a shutter speed value as an integer value without a unit. In this case, the Low-Level API provides two conversion functions between shutter speed and exposure time:

`duShutterSpeedToExposureTime()` and `duExposureTimeToShutterSpeed()`. The unit of exposure time is msec.

Function call:

```
int duShutterSpeedToExposureTime(int shutter, double* exposure_time);
int duExposureTimeToShutterSpeed(double exposure_time, int* shutter);
```

C/C++:

```
int ret_value = 0;
int MCLK = 5;    // 5MHz, master clock of MCU to sensor
int shutter_speed = 100;
double exposure_time = 0;
ret_value = duShutterSpeedToExposureTime(MCLK, shutter_speed,
&exposure_time);

exposure_time = 100;    // 100 msec
shutter_speed = 0;
ret_value = duExposureTimeToShutterSpeed(MCLK, exposure_time,
&shutter_speed);
sdkSetShutterSpeed(shutter_speed);
```

Get Optimal Shutter Speed

Actually, shutter speed is exposure time, that is, the length of time when NSP32 spectral sensor is exposed to input light. The amount of input light that reaches to NSP32 spectral sensor is proportional to the shutter speed or exposure time. To have correct spectra with NSP32 spectral sensor, you must set shutter speed value to have enough amount of light. The value of shutter speed need to be increased or decreased depends on the power of input light of interest. The ADC resolution of nanoLambda's NSP32 spectral sensor is 12-bit, so maximum DN what you can achieve by adjusting shutter speed is 4096. As thumb of rule, higher DN is better spectra. We calls a certain shutter speed as 'optimal' shutter speed with which the closest value to maximum DN(4096) is achieved. Due to the different illumination conditions of your spectral sensing experiments with NSP32 spectral sensor, you may don't know what shutter speed is optimal to acquire best spectra. We provides one convenient function that works based on AE(Auto-Exposure) function:

`duGetOptimalShutterSpeed()`. If you are work in the low light condition, it will take significant time (several tenth seconds) to get 'optimal' shutter speed. If you are work with the too strong light source, NSP32 spectral sensor can be saturated very easily even with minimum shutter speed (1).

`duGetOptimalShutterSpeed()` function normally returns an integer from 1 ~ `NSP_DEVICE_MAX_SS`, indicating the allowable range of shutter speed. If a found shutter speed value is larger than `NSP_DEVICE_MAX_SS`, `NSP_DEVICE_MAX_SS` is returned and this condition means that light quantity on NSP32 spectral sensor is too low. If too many nano-filters are saturated, then this function returns "-1".

Function call:

```
int duGetOptimalShutterSpeed(int* valid_filters, int num_valid_filters);
```

C/C++:

```
int num_valid_filters = duGetNumOfValidFilters();
```



```
int* valid_filters = duGetValidFilters();
int optimal_shutter_speed = duGetOptimalShutterSpeed(valid_filters,
                                                    num_valid_filters);

if( optimal_shutter_speed == -1 ) // filter values are saturated
    printf("Illumination is too strong (saturation is happened.)\n");
else if( optimal_shutter_speed == NSP_DEVICE_MAX_SS ) // 1200
    printf("Illumination is too weak.\n");
else
    printf("Illumination is in normal state.\n");
```

Note: Setup illumination

Illumination can be subject to three conditions:

- 1) it is too dark
- 2) normal
- 3) it is too bright(strong).

You are not guaranteed to get the correct spectrum in 1) or 3). If your lighting condition is 1), you must increase the amount of light reaching the sensor by reducing the distance between the spectrum sensor and the light source, or by increasing the shutter speed value. In case 3), contrary to 1), the amount of light reaching the sensor must be reduced by increasing the distance between the spectrum sensor and the light source or by decreasing the shutter speed value. The requirement for NSP32 spectral sensor to reliably generate the correct spectra is that none of the 1024 nano-filters will saturate under some sort of light source while the response of each nano-filter should approach the maximum (4096). It is very important for users to find the optimum distance between the light source and NSP32 spectral sensor and the optimum shutter speed value. After finding this optimal condition, you can find the fine tuned conditions (distance between the light source and the spectrum sensor, and shutter speed) that can get the best spectrum by slightly increasing or decreasing the value of the shutter speed. This optimal state ensures accurate spectral acquisition.

Get Filter Data

You can obtain nano-filter data directly from NSP32 spectral sensor using `duGetFilterData()` function. NanoLambda has separated the interface API for the device (sensor) and the core API for the spectrum calculation to support the use case for the user (developer) to develop/create the interface to NSP32 spectral sensor using the MCU selected by the user. The user (developer) can acquire raw nano-filter data using the custom device interface developed/created by him/her and convert the filter data into spectrum data using Core API provided by nanoLambda. As mentioned in the part '*Get / Set Sensor Parameters*' in this section, when you directly acquire raw sensor data using the Low-Level API and convert it into spectral data, care must be taken in setting sensor parameters. Sensor parameters, especially ADC gain and range values, must be set not arbitrarily but using the correct information contained in the calibration data file of NSP32 spectral sensor. These sensor parameters can be obtained using the `csGetSensorParameters()` function from the calibration data file.

Note: Effective and monitoring filters

One filter data array consists of 1024 filters in total and consists of two groups, one group of effective(valid) filters and one group of monitoring filters, depending on the design and application purposes of NSP32 spectral sensor. The monitoring filters are only used internally by nanoLambda, and only valid filters are used for actual spectrum calculations. A list of valid filters is provided by nanoLambda, a list common to all NSP32 spectral sensors may be used, and a separate list for each NSP32 spectral sensor or other NSP32 spectral sensor may be used. The list of valid filters for each NSP32 spectral sensor is included in the calibration data file of the corresponding sensor, and this list is used in the spectrum reconstruction process.

Function call:

```
int duGetFilterData(double* output, int num_of_averages);
```

C/C++:

```
int total_num_of_filters = duGetNumOfFilters();
int num_of_averages = 50; // count for frame averaging
double output[total_num_of_filters];
int ret_val = duGetFilterData(output, num_of_averages);
if( ret_value == NSP_RETURN_VALUE_SUCCESS){
    // calculate spectrum data using Core API function
    csCalculateSpectrum(...);
}
```

Get Valid Filters

You can query the number and data array of valid filters that make up one NSP32 spectral sensor if necessary. The function `duGetNumOfValidFilters ()` returns the total number of filters and the function `duGetValidFilters ()` returns an integer array of valid filters. The total number of filters in different versions of NSP32 spectral sensor may vary.

Function call:

```
int duGetNumOfValidFilters(); // returns the number of valid filters
int* duGetValidFilters(); // return the array of valid filters
```

C/C++:

```
int num_valid_filters = duGetNumOfFilters();
int* valid_filters = duGetValidFilters();
if( valid_filters != NULL){
    // control spectral sensor
}
```

Get/Set Sensor Parameters

Normally you will want to set one or more acquisition parameters controlling the conditions under which spectra are obtained. “*Chapter 5: Acquisition Parameter*” lists all available parameters. Once again, three parameters (shutter speed, ADC gain value and ADC range value) are essential parameters for sensor data acquisition and spectra calculation. Except shutter speed, both ADC gain

and ADC range values are queried from sensor calibration data and you must set up those values to NSP32 spectral sensor explicitly when you use Low-Level API functions.

Note: Use sensor parameters from sensor calibration file

With Low-Level device (sensor) control functions, you can change sensor parameters like ADC gain, range, and resolution by yourself. But, every NSP32 spectral sensor delivered with sensor specific calibration data file which was generated under the pre-specified ADC settings. This means that to acquire correct spectra, you must use sensor parameters from sensor calibration file. If you use arbitrarily set of parameters rather than sensor parameters from the sensor calibration file, the suitability and accuracy of the acquired spectra will not be guaranteed.

ADC gain value can be set to 0 (4x) or 1 (1x), ADC range value can be set to any value from 0 to 255, and ADC resolution can be set to 0 (12-bit) or 1. But, nanoLambda delivers NSP32 spectral sensor with fixed values: ADC gain = 1 and ADC resolution = 0. Don't try to set with different values. You will get wrong spectra. ADC range value is a variable depends on NSP32 spectral sensor calibration condition.

Function call:

```
int duGetShutterSpeed (void);
int duSetShutterSpeed (int shutter_speed);
int duGetShutterSpeedLimits (int* min_val, int* max_val);
int duGetSensorParameters(int* adc_gain, int* adc_range, int* adc_res);
int duSetSensorParameters(int adc_gain, int adc_range, int adc_res);
```

C/C++:

```
// query current shutter speed
int old_shutter_speed = 0;
old_shutter_speed = duGetShutterSpeed();

// set shutter speed with new value
int new_shutter_speed = 100;
int res_val = duSetShutterSpeed(new_shutter_speed);

// you can check the available range of shutter speed
int min_shutter_speed = 0;
int max_shutter_speed = 0;
res_val = duGetShutterSpeedLimits(&min_shutter_speed,
&max_shutter_speed);

// get current register settings from spectral sensor
int adc_gain = 0, adc_range = 0;
res_val = duGetSensorParameters(&adc_gain, &adc_range);

// change sensor registers with new values
adc_gain = 1;
adc_range = 246;
res_val = duSetSensorParameters(adc_gain, adc_range);
```

Disconnect NSP32 Spectral Sensor

You can disconnect your NSP32 spectral sensor with `duDisconnect()` function which free all connections to sensors, destroys the sensor interface object, and free allocated system resources internally.

Function call:

```
int duDisconnect(void);
```

C/C++:

```
int ret_val = duDisconnect();
```

Create Core Spectrum Object

To calculate spectra, to query spectral properties of a particular NSP32 spectral sensor, and to retrieve sensor parameters (ADC gain and ADC range), you must first create a Core Spectrum object. The main goal of the Core Spectrum object is to calculate the spectra from the sensor data obtained from the NSP32 spectral sensor. The calibration data specific to each NSP32 spectral sensor is participated in spectra calculations. Core Spectrum object is created by `csCreate()` function and will return `NSP_RETURN_VALUE_SUCCESS` if it succeeded or `NSP_RETURN_VALUE_FAILURE` if it failed.

Note: Singleton SDK object

Your application must create only one instance of SDK object. If you try to create another instance of SDK object, `csCreate()` will return `NSP_RETURN_VALUE_SUCCESS` value without additional creation of this object. All interactions with every nanoLambda NSP32 spectral sensor(s) which are attached to your computer must be performed within this one single executable/application.

Function call:

```
int csCreate(void);
```

C/C++:

```
int ret_val = csCreate();
if( ret_value == NSP_RETURN_VALUE_SUCCESS){
    // control spectral sensor
}
```

Register a Calibration Data to Repository

Once Core Spectrum object is created successfully, next, you must register one sensor calibration data to the sensor data repository by providing a sensor calibration data file path as an argument to `csRegister()` function. If you have a return value larger than 0, then this addition is succeeded and from now on, you can use this sensor for your application by activating it with `csActivateSensorWithID()` function.

The `csRegister()` function normally returns an integer from 0 – N, indicating the number of NSP32 spectral sensors in the sensor data repository which managed by SDK internally. If an error

occur during sensor data registration to the repository, this method will return “0” or negative value as exception code.

Function call:

```
int csRegister (std::string sensor_cal_data_path);
```

C/C++:

```
std::string sensor_cal_path = "./config/sensor_Y8585-1-85-85-0.dat";
int total_count_of_active_sensors = csRegister(sensor_cal_path);
if( total_count_of_active_sensors > 0){
    // do something..
}
```

Activate Calibration Sensor Data

To use a specific sensor calibration data with a corresponding physical NSP32 spectral sensor, user must activate a sensor calibration data first by calling `csActivateSensorWithID()` function with sensor ID string as an input argument. If there are multiple NSP32 spectral sensors in your system, this is only way to switch between different sensor calibration data and to use one calibration data to calculate spectra.

The function `csActivateSensorWithID()` normally returns an integer `NSP_RETURN_VALUE_SUCCESS` or `NSP_RETURN_VALUE_FAILURE` indicating success or failure, respectively.

Function call:

```
int csActivateSensorWithID (const char* sensor_id_str);
```

C/C++:

```
const char* sensor_id_str = "Y8585-1-85-85-0";
int ret_val = 0;
ret_value = csActivateSensorWithID(sensor_id_str); //activate with ID
if( ret_value == NSP_RETURN_VALUE_SUCCESS ){
    // do something..
}
```

Get Spectral Information

Four different types of sensors (VIS, NIR1, NIR2 and FULL) of the nanoLambda are shipped to the user with their own calibration data file, each calibrated for a different target wavelength range. Therefore, you can use `csGetWavelengthInfo()` and `csGetResolution()` functions to retrieve information about the target wavelength range and resolution of your NSP32 spectral sensor. This information is the information that matches the wavelength data returned by `csCalculateSpectrum()`.

Function call:

```
int csGetWavelengthInfo(double *start_wavelength, double
*end_wavelength, double *interval_wavelength);
int csGetResolution (double* resolution);
```

C/C++:

```
int ret_value = 0;
double start_wavelength = 0.0;
double end_wavelength = 0.0;
double interval_wavelength = 0.0;
double resolution = 0.0;
ret_value = csGetWavelengthInfo(&start_wavelength, &end_wavelength,
                                &interval_wavelength);
ret_value = csGetResolution(&resolution);
```

Get Sensor ID from Calibration File

If you wish to identify which NSP32 spectral sensor calibration data is currently activated in the repository and you are using to calculate spectra, there is a Low-Level function that helps identify NSP32 spectral sensor by returning sensor's unique ID. You can get a sensor ID from NSP32 spectral sensor calibration file with `csGetSensorID()` function. You must always keep in mind that the physical sensor ID and the ID from the calibration file must be matched before start spectrum calculations.

Function call:

```
int csGetSensorID(const char* sensor_id_str);
```

C/C++:

```
char sensor_id[SENSOR_ID_STRING_LENGTH];
int ret_value = 0;
ret_value = csGetSensorID(sensor_id);
```

Get Capacity and Sensor List From Repository

You can use the `csGetSensorList()` function to get whole sensor IDs as an array of character array pointer from your NSP32 spectral sensor calibration data repository that you added calibration file by `csRegister()` function explicitly and managed by Core Spectrum object internally. This function will allocate the necessary memory to store whole sensor IDs(string). To allocate memory space for sensor IDs (strings), you must know about how many sensors are in your sensor data repository in advance. You can get total sensors in sensor data repository by calling `csCapacity()`.

Note: Get a list of all sensors in your sensor data repository

You can use the `csGetSensorList()` function to get a list of sensor IDs that exist in the repository of NSP32 spectral sensor calibration data that you explicitly added using the `csRegister()` function. Do not confuse the `duGetSensorList()` function, which returns the ID list of one or more NSP32 spectral sensors connected to your system.

Function call:

```
int csCapacity (void);
char**csGetSensorList ();
```

C/C++:

```
int capacity = csCapacity();
char** sensor_list_from_DB = NULL;
sensor_list_from_DB = csGetSensorList();
```

Get Sensor Parameters From Cal File

Normally you will want to get sensor parameters from sensor calibration file to set a NSP32 spectral sensor's registers. Once again, three parameters (shutter speed, ADC gain value and ADC range value) are essential parameters for sensor data acquisition. Among them, ADC gain and ADC range values from sensor calibration file must be set to the registers in NSP32 spectral sensor before you try to acquire sensor data and calculate spectra.

Note: Use sensor parameters from sensor calibration file

With Low-Level device (sensor) control functions, you can change sensor parameters like ADC gain, range, and resolution by yourself. But, every NSP32 spectral sensor delivered with sensor specific calibration data file which was generated under the pre-specified ADC settings. This means that to acquire correct spectra, **you must use sensor parameters from sensor calibration file to set NSP32 spectral sensor**. If you use arbitrarily set parameters rather than sensor parameters from the sensor calibration file, the suitability and accuracy of the acquired spectra will not be guaranteed.

nanoLambda delivers NSP32 spectral sensor with fixed values: ADC gain = 1 and ADC resolution = 0. So, you will always get those numbers from sensor calibration data. In case of ADC range, it will be variable according to the sensor calibration conditions of nanoLambda.

Function call:

```
int csGetSensorParameters(int* adc_gain, int* adc_range);
```

C/C++:

```
// query sensor parameters from calibration file
int adc_gain = 0, adc_range = 0;
res_val = csGetRegisters(&adc_gain, &adc_range);

// set sensor parameters to physical device(sensor)
res_val = duSetSensorParameters(adc_gain, adc_range);
```

Set Background Filter Data

NSP32 spectral sensor of nanoLambda has a non-zero dark current in the absence of illumination. This dark current and ambient noise must be corrected before performing spectral calculation. To perform this correction, you must acquire sensor data at shutter speed 1 and then use the `csSetBackground()` function to specify this sensor data as background data explicitly. If the sensor data for background is acquired at SS > 1 and/or don't be specified as the background data, the

accuracy and validity of the final spectrum is not guaranteed.

Note: Background sensor data acquisition at shutter speed 1

A sensor data as background data must be acquired always at shutter speed 1. If not, final spectrum data will not accurate and valid.

Function call:

```
int csSetBackground(double* background_filter_data);
```

C/C++:

```
#include "nsp_device_api.h"
#include "nsp_core_api.h"
#include "nsp_core_spectrum_common.h"

int ret_val = 0;
int shutter_speed = 1;
int frame_averages = 50;
double filter_data[1024];

//// Background filter data must be acquired with shutter speed 1.
// set shutter speed
ret_val = duSetShutterSpeed(shutter_speed);

// get raw sensor data
ret_val = duGetFilterData(filter_data, frame_averages);

// set raw sensor data as background data
ret_val = csSetBackground(filter_data);
```

Acquire a Spectral

Now you are ready to acquire a spectrum. A spectrum is simply a 1-D double type array of calibrated response values of NSP32 spectral sensor. The Low-Level function to compute the spectrum is `csCalculateSpectrum()`. When you call the `csCalculateSpectrum()` function, this function will calculate a spectra with a sensor data and a sensor calibration data, and then will return the spectral and the wavelength data to your application. Before calculate one spectra with one sensor data, you must know about the length of spectra with `csGetSpectrumLength()` function to allocate memory buffers to store spectra and wavelength data.

Function call:

```
int csGetSpectrumLength(void);
int csCalculateSpectrum(double* filter_data_in, int shutter_speed,
double* spectrum_out, double* wavelength_out);
```

C/C++:

```
#include "nsp_device_api.h"
#include "nsp_core_api.h"
#include "nsp_core_spectrum_common.h"
```



```
int ret_val = 0;
int shutter_speed = 30;
int frame_averages = 50;
double filter_data[1024];
double background_filter_data[1024];

// set shutter speed to acquire background data
ret_val = duSetShutterSpeed(1);

// acquirebackground raw sensor data
Ret_val = duGetFilterData(background_filter_data, frame_averages);

// set raw sensor data as background data
ret_val = csSetBackground(background_filter_data);

// set shutter speed
ret_val = duSetShutterSpeed(shutter_speed);

// acquireraw sensor data
Ret_val = duGetFilterData(filter_data, frame_averages);

// get spectrum and wavelength data
double* spectrum_out = NULL;
double* wavelength_out = NULL;
int spectrum_size = sdkGetSpectrumLength();

// allocate memory for spectrum and wavelength data
spectrum_out = (double *)malloc(spectrum_size*sizeof(double));
wavelength_out = (double *)malloc(spectrum_size*sizeof(double));
if( spectrum_out && wavelength_out ){
    // calculate spectrum
    ret_val = csCalculateSpectrum(filter_data, shutter_speed,
                                spectrum_out, wavelength_out);
    // ...
}
if( spectrum_out ) free((void *)spectrum_out);
if( wavelength_out ) free((void *)wavelength_out);
```

Destroy Sensor Interface

When your application is ready to terminate, call the `csDestroy()` function to destroy Core Spectrum object.

Function call:

```
int csDestroy(void);
```

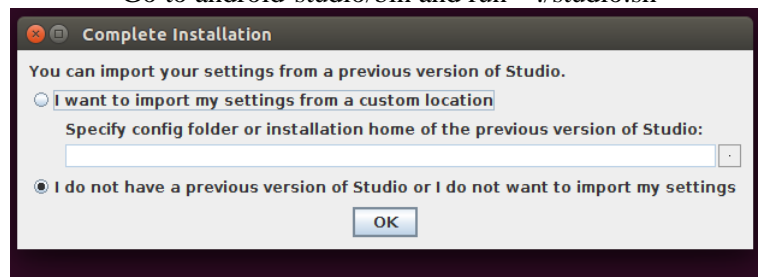
Chapter 8 Developing Your NSP32 SDK Application

Overview

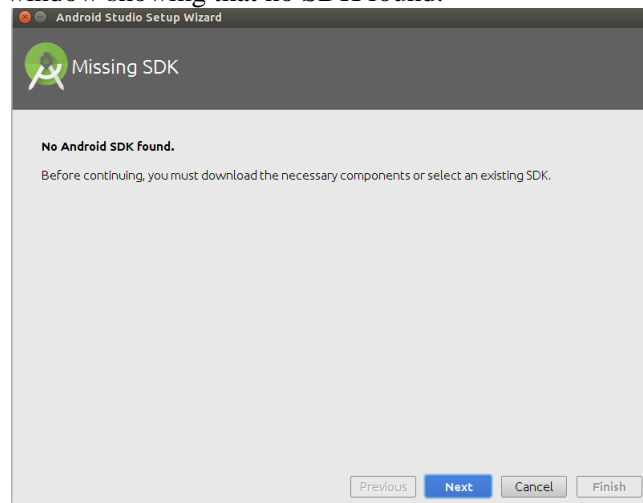
The following sections provide information on setting up the development environment for your specific programming language to develop NSP32 SDK applications.

Android Studio Setup on Linux

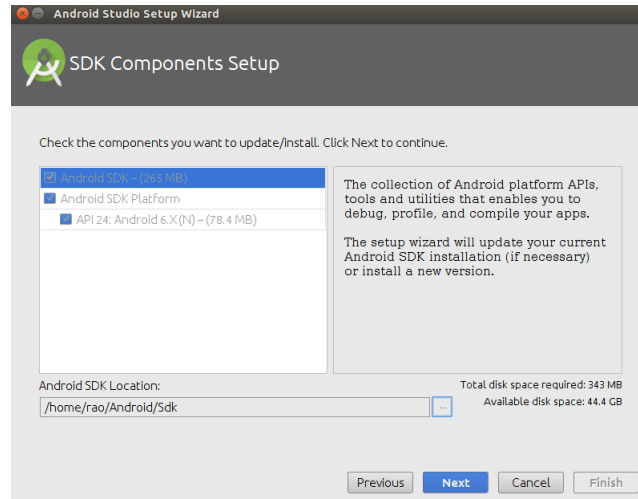
1. Installing JDK .18.0_91
 - `sudo add-apt-repository ppa:webupd8team/java`
 - `$ sudo apt-get update`
 - `$ sudo apt-get install oracle-java8-installer`
 - `$ java -version`
 - `$ sudo apt-get install oracle-java8-set-default`
2. Installing 32bit libraries
 - `$ sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 lib32stdc++6`
3. Install Android Studio: <https://developer.android.com/studio/index.html>
 - Extract the files.
4. Run Android Studio
 - Go to `android-studio/bin` and run `./studio.sh`



- click 'OK'. Android studio will start
- You will get one window showing that no SDK found.



- Click Next. Here you can change the path of SDK but it is recommended that you use the same path.



- Click Next and then Finish and it will download the SDK.
5. Installing the NDK
 - You can follow below mentioned link to download NDK.
 - <http://developer.android.com/sdk/ndk/index.html>
 - Download the NDK package for the development environment you use
 - Extract the package to a directory.
 6. Installing standalone Tool-chain:
 - `/path/to/NDK/build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-21 --install-dir=/tmp/my-android-toolchain`
 - For reference: https://developer.android.com/ndk/guides/standalone_toolchain.html

Developing in C/C++ (all IDEs)

Required Header Files

If you are developing in C or C++, you will need several header files.

You must update your IDE's project settings by adding the following directories to the list of include directories:

```
$(NSP32_SDK_HOME)\include\
$(NSP32_SDK_HOME)\include\base
$(NSP32_SDK_HOME)\include\core
$(NSP32_SDK_HOME)\include\device
```

NSP32 ADK_HOME refers to the location where NSP32 SDK was installed.

Note:

The included header files would be different for your applications. Please refer to 'examples' files. For example, if you're developing your application/prototype with NSP32 ADK, header files at below are need to be included:

- `nsp_core_common.h`
- `nsp_device_api.h`

- `nsp_core_api.h`

Required Library Files

You must also update your IDE's project settings for it to locate the NSP32 SDK static libraries: CrystalBase.lib, CrystalCore.lib, and CrystalPort.lib files if you use Low-Level API. If you use High-Level API, then it's enough to include CrystalBox.lib. You also need to have two 3rd party libraries (libusb and libgsl).

You must update your IDE's project settings by adding the following directories to the list of include directories for your target platforms:

<code>\$(NSP32_SDK_HOME)\lib\android</code>	// for Android
<code>\$(NSP32_SDK_HOME)\lib\raspbian-jesse</code>	// for Raspbian-jesse
<code>\$(NSP32_SDK_HOME)\lib\win32</code>	// for Windows 32-bit
<code>\$(NSP32_SDK_HOME)\lib\win64</code>	// for Windows 64-bit
<code>\$(NSP32_SDK_HOME)\lib\ubuntu</code>	// for Ubuntu
<code>\$(NSP32_SDK_HOME)\lib\macOS</code>	// for Mac OS-X

- Low-level APIs: CrystalBase.lib, CrystalCore.lib, CrystalPort.lib
- High-level API: CrystalBox.lib

3rd Party Requirements

If you are developing your application on NSP32 SDK supported platforms like Android, Raspbian-jesse, Linux, and Mac OS, to acquire NSP32 sensor data through USB, you must download and install two open source libraries: libgsl and libusb libraries. The version 1.0.9 or higher of libusb acceptable.

Download the libusb from <http://libusb.info/> and libgsl from <https://www.gnu.org/software/gsl/>.

Specify Your OS Platform

It may be necessary to add one of the following #define lines to your source files:

```
#define __WIN32__ (use this for Windows 32-bit and 64-bit versions)
#define __MAC__
#define __LINUX__
```

Choose the symbol that corresponds to the operating system you are using. This #define should be placed BEFORE any include statements for NSP32 SDK header files. Some IDE's and project types automatically define the appropriate symbol. But some types of Visual Studio projects do not automatically define the required symbol. If none of these symbols is defined, you will get numerous compile-time errors relating to NSP32 SDK symbols and functions.

Developing with Microsoft Visual C++ 10.0

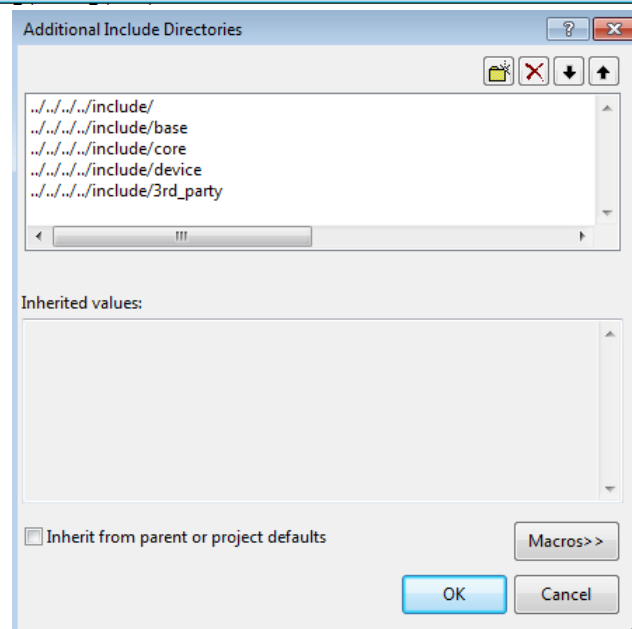
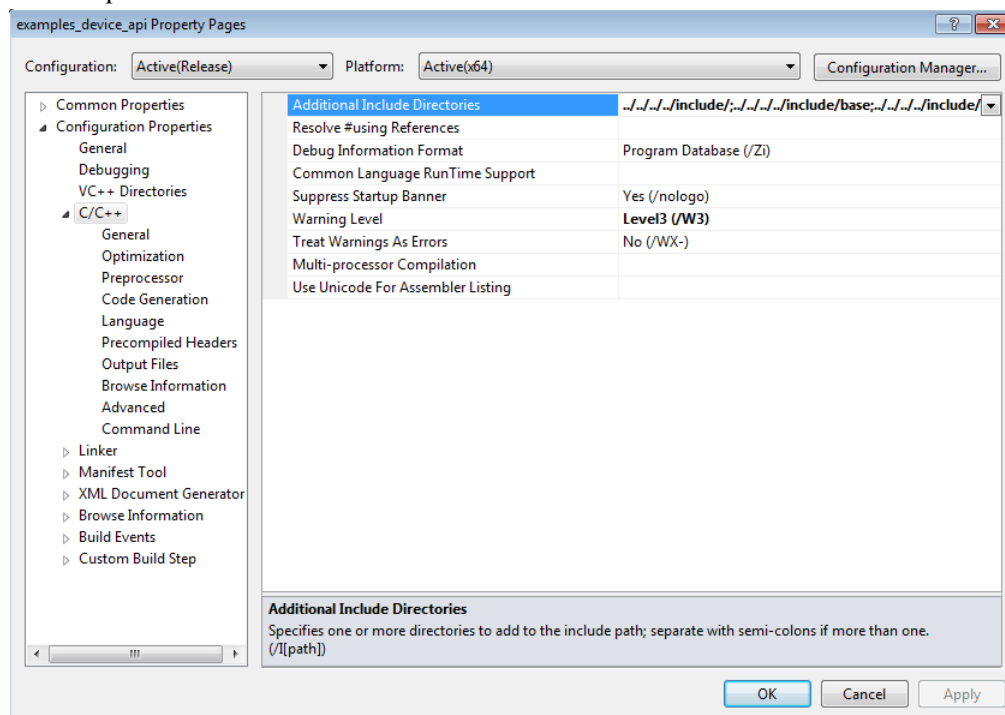
Specifying the Location of the C/C++ Header Files

You must specify where to find the C/C++ header files.

► Procedure

To set the path where the IDE will look for the C/C++ header files:

1. Click on the 'Project'→'Properties' menu item.
2. Select 'All Configurations' from the 'Configuration' drop down list box.
3. Expand the 'Configuration Properties' in the pane on the left side.
4. Click on the 'C/C++' item in the pane.
5. Add the path for C/C++ header files to 'Additional Include Directories'.



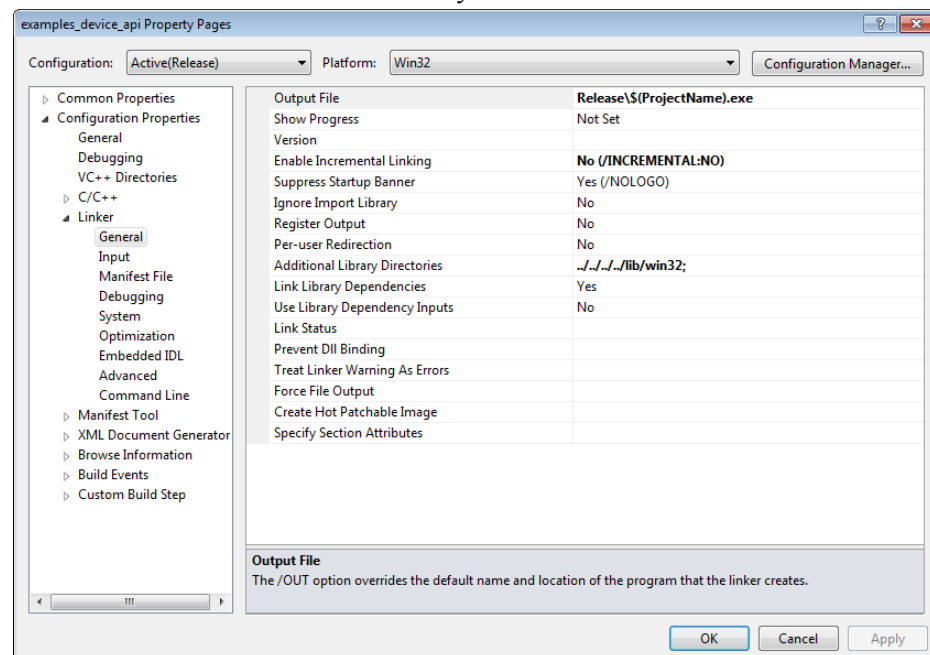
Specifying the Location of the NSP32 SDK DLLs

You must specify where to find the NSP32 SDK DLLs(CrystalBase, CrystalPort, CrystalCore, and CrystalBox).

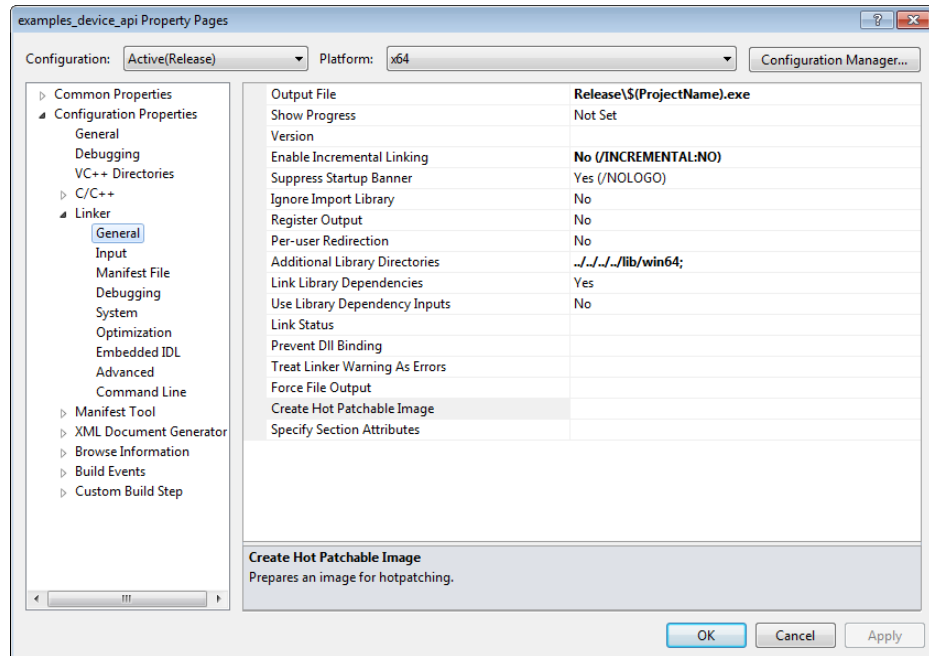
► Procedure

To set the path where the IDE will look for the NSP32 SDK's DLLs:

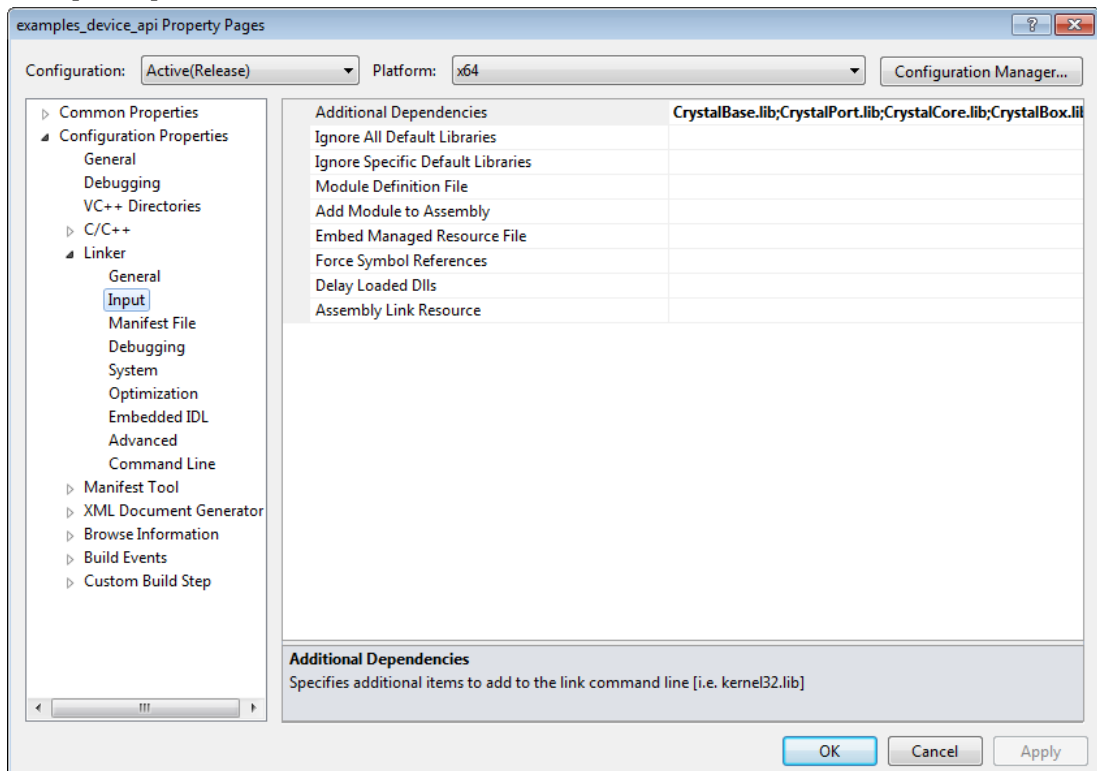
1. Click on the 'Project'→'Properties' menu item.
2. Select 'All Configurations' from the 'Configuration' drop down list box.
3. Expand on the 'Configuration Properties' in the pane on the left side.
4. Click on the 'Linker' item in the pane.
5. Add the path for DLL files to 'Additional Library Directories' in 'General' item.
6. Add the DLL files names to 'Additional Library Directories' in 'General' item.



[32-bit]



[64-bit]



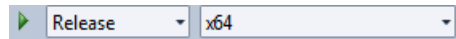
Modifying Your C++ Project Settings for 64-bit Windows

The examples that come with NSP32 SDK are initially set to generate 32-bit Windows executable(EXE). The following shows you how to modify the project settings for the examples to generate 64-bit executable (EXE).

Create an x64 Platform Option

► Procedure

1. Choose 'x64' from the solution platform drop-down menu at the top of the Visual Studio 2010 or higher GUI if it exists.



2. If not
3. Choose 'Configuration Manager' from the Solution Platforms drop-down menu at the top of the Visual Studio 2010 or higher GUI.
4. Choose <New...> from the Active solution platform drop-down menu.
5. Choose x64 from the 'Type' or select 'the New Platform' drop-down menu.
6. Choose Win32 from the Copy settings from drop-down menu.
7. Check Create new project platforms.
8. Make sure the x64 platform is selected when you build your application.

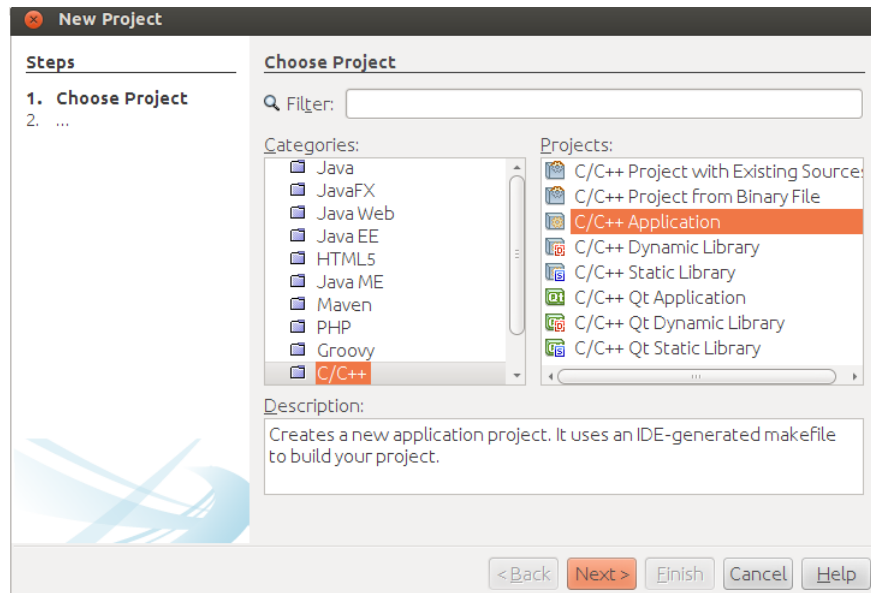
Change Your Library Dependencies

Change library path to : [NSP32_SDK_HOME]/lib/win64

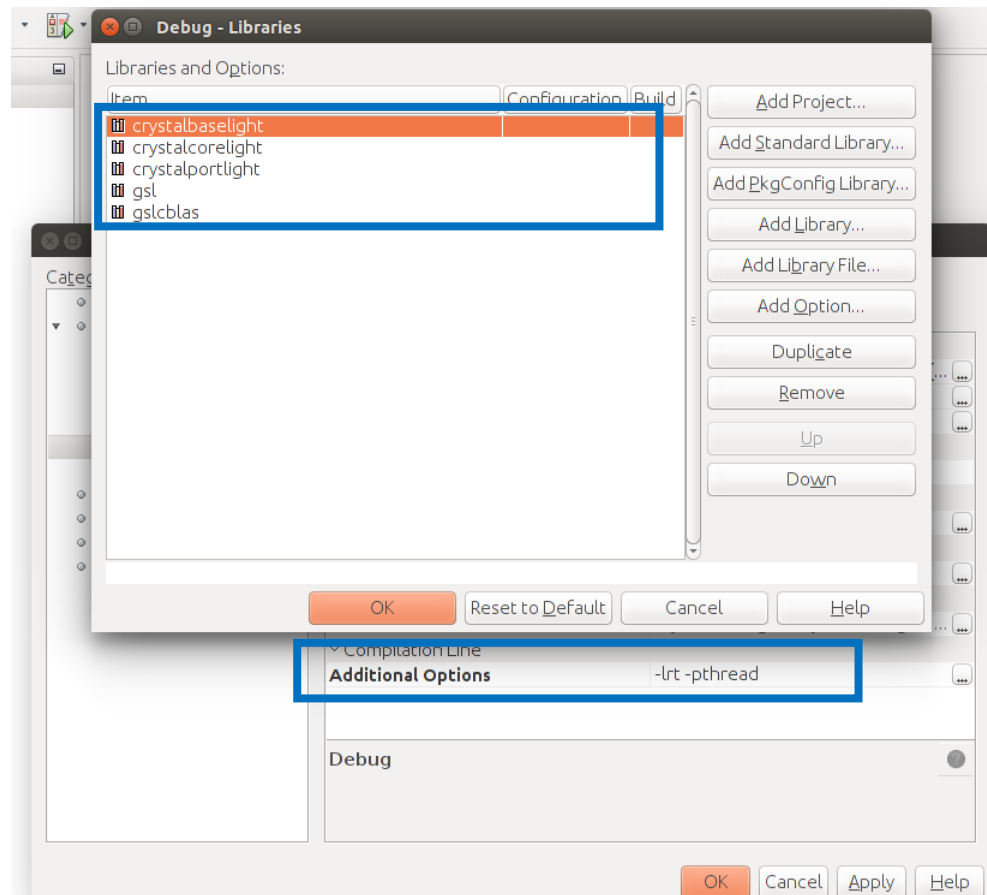
NetBeans Setup on Linux

If you are using the NetBeans IDE on Linux, use the following guidelines for setting up your environment.

1. Install the latest version of NSP32 SDK on your Linux system. Be sure to choose the correct architecture (i.e., 32-bit or 64-bit). For this example, assume your system is 64-bit architecture, so the installer would be similar to the following: *NSP32 SDK-1.7-linux64-development-installer.tar.gz*
2. This procedure assumes you install NSP32 SDK into your home directory. In NetBeans, open the sample project located in
`"~/nanoLambda/NSP32 SDK-1.7/examples/cpp/netbeans/ubuntu
 /examples_device_spectrum_api/nbproject".`
3. Now, you are ready for build examples from nanoLambda or your own code. For building and running console based example, we will again use the same NetBeans IDE and this time we will select "C/C++ Application" projecttype from the "New Project" dialog window:



4. On the next screen followed by “*Next*” button click, enters the project name and project location and click “*Finish*”.
5. After this, go to the project on NetBeans IDE and right click and select “*Properties*” on pop-up menu.
6. Change the C Compiler, C++ Compiler, and Assembler and Linker under “*Build*” property:
 - C Compiler: gcc (default)
 - C++ Compiler: g++ (default)
 - Assembler: as (default)
 - Linker : g++ (default)
7. In the last, you must add two more things to “*Linker*”:
 - Libraries: crystalbase.a, crystalcore.a, and crystalport.a.
 - You can add 3 libraries included in NSP32 SDK to “*Libraries*” field by using “*Add Library...*” on the “*Libraries*” dialog window.
 - One additional options : -lrt -pthread
 - You can add this option to “*Additional Options*” field as below.



8. Build and run the **example** application to verify your environment has been set up correctly.

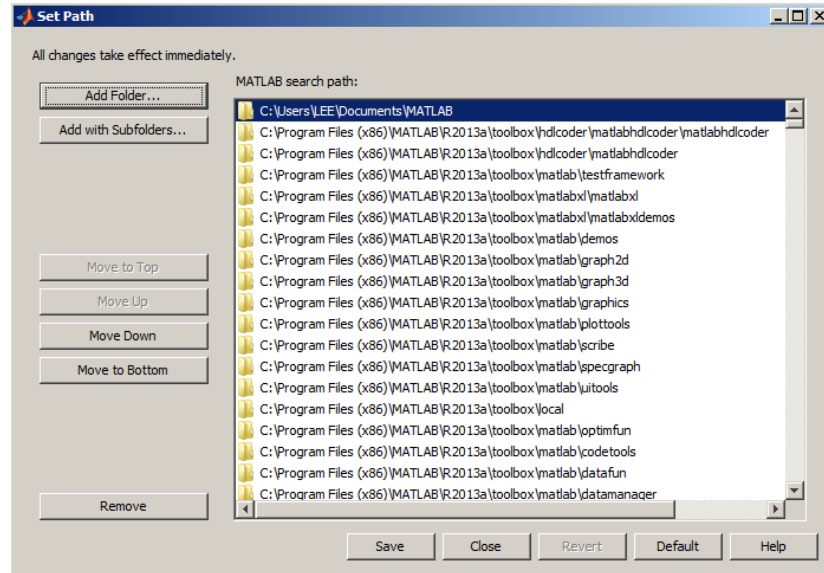
Developing in MATLAB

The following sections provide information on how to write your MATLAB scripts with MATLAB wrappers to access NSP32 SDK functionality

Set Path to MATAB Wrapper

► Procedure

1. Start MATLAB and click 'Set Path' at 'Home' tab.
2. Click 'Add Folder...' and select '[NSP32_SDK_HOME]\wrappers\matlab\' folder.



Write Your MATLAB Script

The following procedure assumes you are writing your own test scripts in MATLAB workspace.

- 32-bit version (Matlab and NSP32 SDK DLLs)
`LIBRARY_PATH = '\lib_win32\';`
- 64-bit version (Matlab and NSP32 SDK DLLs)
`LIBRARY_PATH = '\lib_win64\';`

```
CORE_LIBRARY_NAME = 'CrystalCore.dll';
DEVICE_LIBRARY_NAME = 'CrystalPort.dll';
```

```
path(path, [cd , '/../..wrappers/matlab']);
path(path, [cd , LIBRARY_PATH]);
```

► Procedure

1. Add a folder containing DLLs to your paths


```
LIBRARY_PATH = '/lib_win32/';
path(path, [cd , '/../..wrappers/matlab']);
path(path, [cd , LIBRARY_PATH]);
```
2. Load two NSP32 SDK DLLs (CrystalCore.dll, CrystalPort.dll) explicitly


```
CORE_LIBRARY_NAME = 'CrystalCore.dll';
DEVICE_LIBRARY_NAME = 'CrystalPort.dll';
[ret_device] = load_device_api_library([cd LIBRARY_PATH
DEVICE_LIBRARY_NAME]);
[ret_core] = load_core_api_library([cd LIBRARY_PATH
CORE_LIBRARY_NAME]);
```
3. Write your code to control NSP32 spectral sensor and get spectra.
 - You can find an MATLAB example code from '*Appendix D: NSP32 SDK Examples*'.
4. Refer to MATLAB examples: [NSP32_SDK_HOME]\examples\matlab\'

Developing in LabVIEW

NSP32 SDK now provides C/C++ interface. The directory path '[NSP32_SDK_HOME]\wrappers\labview\Libs' contains the *.dll libraries from the NSP32 SDK DLL files. You can develop your application with two separated parts: 1) 'initialization' and 2) 'main procesing'.

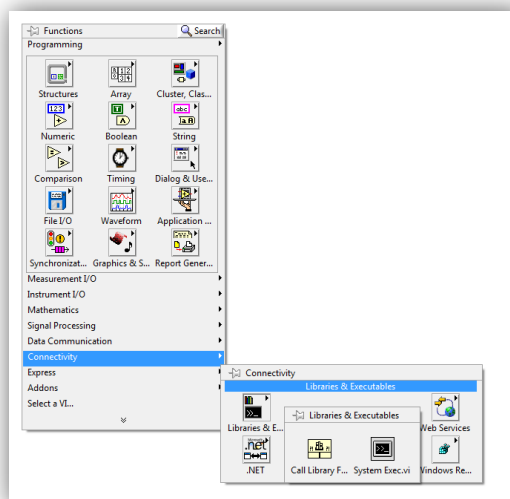
In '**initialization**' part, some of the important functions used are:


- 1) Connecting to the physical NSP32 device (CrystalPort.dll)
- 2) Creating a core open source object. (CrystalCore.dll)
- 3) Loading the sensor data file and registering it. (CrystalCore.dll)
- 4) Getting the sensor Id from the physical NSDP32 device (CrystalPort.dll)
- 5) Getting the sensor Id from the sensor data file (CrystalCore.dll)
- 6) Getting sensor parameters (ADC gain and ADC range) from the sensor data file (CrystalCore.dll)
- 7) Getting sensor parameters (ADC gain and ADC range) from the NSP32 device (CrystalPort.dll)
- 8) Setting sensor parameters (ADC gain and ADC range) to physical NSP32 device (CrystalPort.dll)
- 9) Activating NSP32 physical device with the index. Default is 0. (CrystalPort.dll)
- 10) Acquiring background data with shutter speed 1. (CrystalPort.dll)
- 11) Setting background data. (CrystalCore.dll)

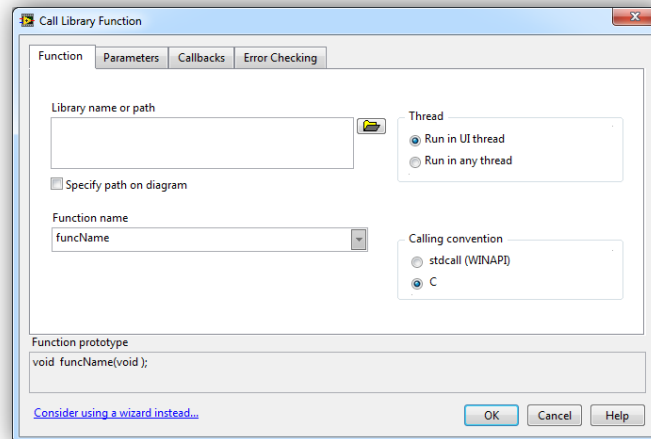
In '**main processing**' part, some of the important functions used are

- 1) Finding optimal shutter speed with AE (CrystalPort.dll)
- 2) Acquiring the Filter data at specific shutter from NSP32 device (CrystalPort.dll)
- 3) Calculating the spectrum data from the earlier acquired filter data. (CrystalCore.dll)
- 4) Find Spectrum Size (CrystalCore.dll)
- 5) Find the spectral information of the specific NSDP32 device (CrystalCore.dll)
- 6) Disconnecting the NSP32 device (CrystalPort.dll)

Right click on the back panel and then select "*Connectivity->Libraries & Executable->Call Library Function (CLF)*"

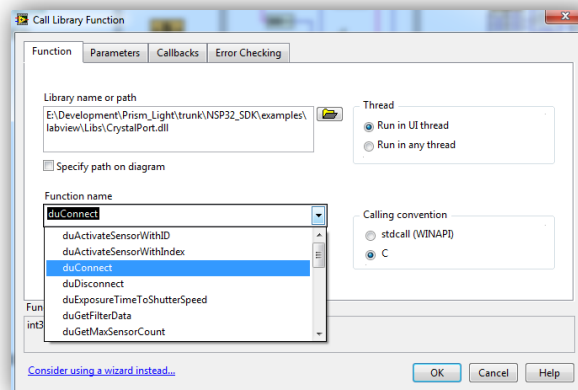


One VI  will be selected. Double click on it.

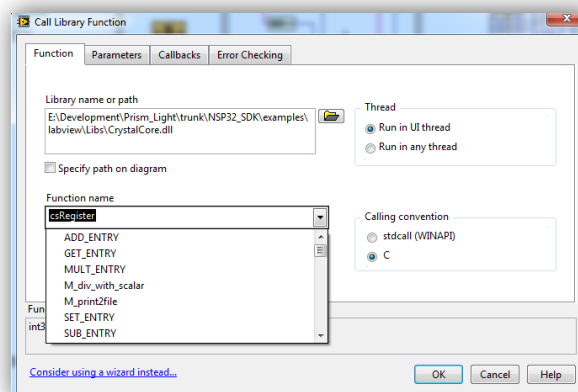


Select the path of the "CrystalPort.dll" or "CrystalCore.dll" depending on the function you want to use. Please make sure all other DLLs such as "CrystalBase.dll" and other are also in the same path. After select "CrystalPort.dll" or "CrystalCore.dll", all the available functions for interface with LabVIEW will be appear in the Drop down list of 'Function name' like this.

For CrystalPort.dll,

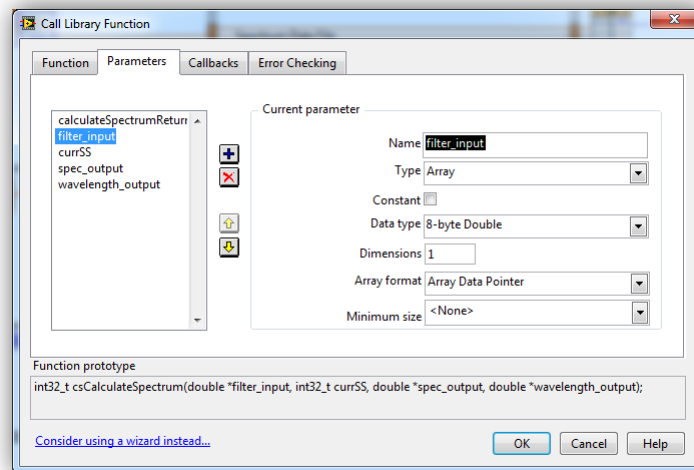


For CrystalCore.dll,

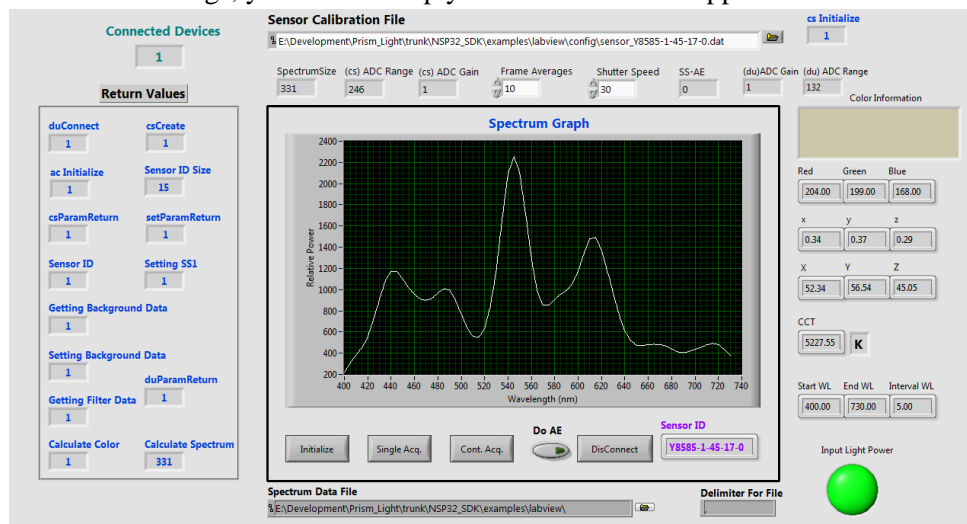


Select the specific function and click 'Parameters' tab from the top. For example, let's select the function "csCalculateSpectrum". After selecting the function, open the header file ('nsp_spectrum_api.h') from the NSP32_SDK/include folder and refer to that function's

declaration, and set the correct return and arguments parameters in ‘Current parameter’ section manually.



After these settings, you can develop your own LabVIEW application as below:



Chapter 9 Example Programs

Overview

This section contains NSP32 SDK example programs for the following 4 representative programming languages:

- **C/ C++**
- **MATLAB**
- **Python**
- **LabVIEW**

C/C++ Examples

```
examples_device_api.exe
examples_spectrum_api.exe
examples_sdk_device_spectrum.exe
```

1. Location:
 - [NSP32_SDK_HOME]\bin\win32\cpp\
2. Language:
 - C/C++, Microsoft Visual Studio 2010
3. Access method:
 - Uses the C/C++ interface to access NSP32_SDK_HOME functionality.
4. API:
 - Low-Level and High-Level
5. GUI:
 - No
6. Device:
 - NSP32ADK
7. Files ([NSP32_SDK_HOME]\examples\cpp\win32_vs2010\):

Low-Level API

- examples_device_api.cpp
- examples_device_spectrum_api.cpp

High-Level API

- examples_sdk_device_spectrum.cpp

8. Demonstrates how to do the the following:

- Connect a NSP32 spectral sensor
- Register a sensor calibration file

- Get NSP32 spectral sensor ID and spectral capability of a sensor
- Set/get acquisition parameters (shutter speed, ADC gain and range, etc.)
- Use AE function to have optimal shutter speed
- Acquire a sensor (filter) data
- Calculate spectra

MATLAB Examples

When you use MATLAB examples, you must make sure that the path to the MATLAB wrappers is set correctly. The location of the MATLAB wrapper is '[NSP32 ADK]\api\Wrappers\Matlab\'. You also must specify the appropriate NSP32 SDK path (win32 or win64) depending on the MATLAB version you are using.

```
spectrometer_example
spectrometer_example_color
spectrometer_example_gui
```

1. Location:
 - [NSP32_SDK_HOME]\examples\matlab\
2. Language:
 - Matlab (2013a or higher, 32-bit and 64-bit)
3. Access method:
 - Uses the MATLAB interface to access NSP32 SDK functionality.
4. API:
 - Wrapper ([NSP_SDK_HOME]\wrappers\matlab\)
5. GUI:
 - No, but with plots: spectrometer_example, spectrometer_example_color
 - Yes : spectrometer_example_gui
6. Device:
 - NSP32ADK
7. Files
 - spectrometer_example.m
 - spectrometer_example_color.m
 - spectrometer_example_gui.m
8. Demonstrates how to do the following:
 - Load/unload API libraries (CrystalPort and CrystalCore)
 - Connect/disconnect a NSP32 spectral sensor
 - Register a sensor calibration file
 - Get NSP32 spectral sensor ID and spectral capability of a sensor
 - Set/get acquisition parameters (shutter speed, ADC gain and range, etc.)
 - Acquire a sensor (filter) data
 - Use AE function to have optimal shutter speed
 - Acquire spectra

LabVIEW Examples

When you start the LabVIEW examples, you must inform the project of the location of your NSP32 SDK DLLs (CrystalBase.dll, CrystalCore.dll and CrystalPort.dll) files and the libraries provided by NSP32 SDK (e.g., CrystalBase.lib, CrystalCore.lib, CrystalPort.lib). For each file click on Browse... and select the files or library function.

example_labview.vi

1. Location:
 - [NSP32_SDK_HOME]\examples\labview\
2. Language:
 - LabVIEW 7.1 and 8.5
3. Access method:
 - LabVIEW wrappers provided by NSP32 SDK
4. API:
 - Wrapper
5. GUI:
 - Yes
6. Device:
 - NSP32ADK
7. Demonstrates how to do the following:
 - Access a NSP32 spectral sensor
 - Specify sensor calibration data file
 - Set acquisition parameters (integration time, etc.)
 - Acquire a spectral (single shot or continuous)
 - Display the spectral in a simple

PythonExamples

example_wrapper_python.py

example_wrapper_python_color.py

1. Location:
 - [NSP32_SDK_HOME]\examples\python\win32\example_wrapper_python
 - [NSP32_SDK_HOME]\examples\python\ubuntu\example_wrapper_python
 - [NSP32_SDK_HOME]\examples\python\raspbian-jessie\example_wrapper_python
2. Language:
 - Python2.x and Python 3.x
3. Access method:
 - Accesses NSP32 SDK ® functionality from python wrapper files
4. API:
 - Wrapper ([NSP32_SDK_HOME]\wrappers\python)
5. GUI:

- No (Console based only)
6. Device:
 - NSP32ADK
 7. Files
 - `example_wrapper_python.py`
 - `example_wrapper_python_color.py`
 8. Demonstrates how to do the following:
 - Load/unload API libraries (CrystalPort and CrystalCore)
 - Connect/disconnect a NSP32 spectral sensor
 - Register a sensor calibration file
 - Get NSP32 spectral sensor ID and spectral capability of a sensor
 - Set/get acquisition parameters (shutter speed, ADC gain and range, etc.)
 - Acquire a sensor (filter) data
 - Use AE function to have optimal shutter speed
 - Acquire spectra
 9. Notes:

These samples should be run from within the Python GUI (IDLE).

Appendix A How to Validate Your Raw Sensor Data?

You can develop the sensor interface board directly based on the sensor specification and hardware manual provided by nanoLambda. Using your sensor board, you can acquire 'raw' sensor data from the NSP32 spectral sensor. The sensor data obtained based on the specific sensor parameters setting is an essential input for spectra calculation using Low-Level API. In this case, you must carefully verify the validity of the sensor data. This is because accurate sensor data can not be guaranteed if sensor data is lost or damaged due to timing or synchronization problems. Only sensor data with no defect will provide accurate spectra.

First of all, you will get 2048 char array data from your sensor interface board. This is the 'raw' sensor data of the NSP32 spectral sensor. The horizontal and vertical sizes of the NSP32 spectral sensor are 32 respectively and the ADC resolution is 12-bit. Therefore, the response of one pixel or filter is output in 2 bytes. For visualization for validation of sensor data or spectra calculation, the 2048 'raw' sensor data must be converted to a 1024 size double type array. You can do this conversion using the code below. The converted 1024 size double type array is input to NSP32 ADK's Low-Level API for spectra calculation. If you enter a 2048 size char array to APIs to calculate spectra, an critical error will occur and you will not get the spectra you want.

```
void convert_2048C_to_1024F(char* inputC, double* outputF)
{
    if( inputC == NULL || outputF == NULL ) return;

    unsignedchar l_framebuf2048[2048];
    memset(l_framebuf2048, 0, sizeof(unsignedchar)*2048);
    memcpy(l_framebuf2048, inputC, sizeof(unsignedchar)*2048);
    memset(inputC, 0, sizeof(unsignedchar)*2048);


    for(int i=0;i<1024;i++){
        outputF[i] = (l_framebuf2048[i*2] << 4) | (l_framebuf2048[i*2+1] >> 4);
    }
}
```

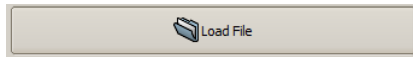
In this chapter we will focus on the question of how to determine if your 'raw' or 'converted' sensor data is valid. Validation of 'raw' or 'converted' sensor data is possible in a visual way, so you can use the simple GUI application tool (SensorIDQ) provided by nanoLambda with the NSP32 ADK for your purpose. The "SensorIDQ.exe" file is located in the "[root directory] \ platforms \ [platforms] \ tools" folder where the NSP32 ADK is installed. For example, on the Win32 platform, you can find the corresponding executable in "[root directory] \ platforms \ Win32 \ tools".

You can use this GUI tool in two ways:

1. **Load the sensor data file with the GUI tool:**

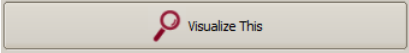
In this case, you can use raw sensor data (2048 bytes char array) obtained from your sensor interface board or a 'converted' sensor data(1024 size double array), and then you should save

the sensor data as a common separated text file (*.txt or *.csv). In more detail, you can visualize sensor data as a gray image by loading (using  or

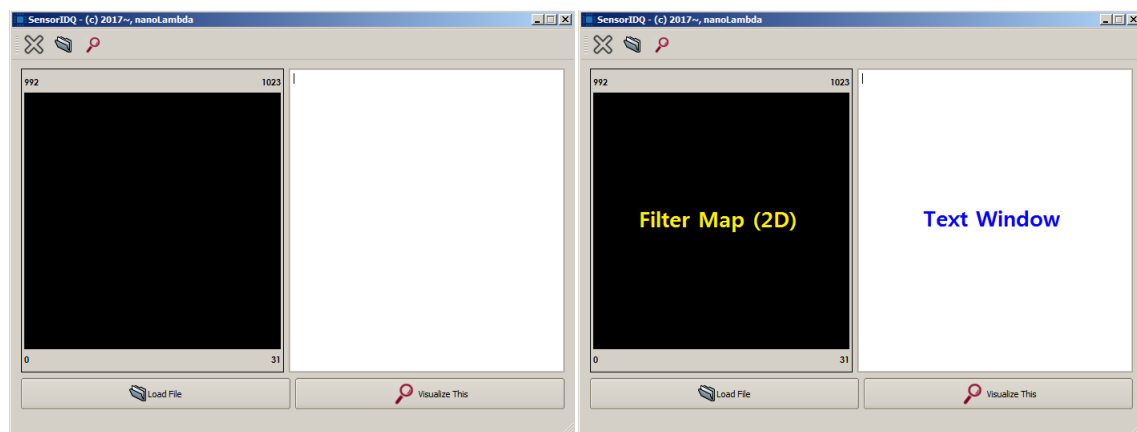


buttons) 2048 size or 1024 size sensor data stored in a text file.

2. Copy the sensor data directly to the GUI tool:

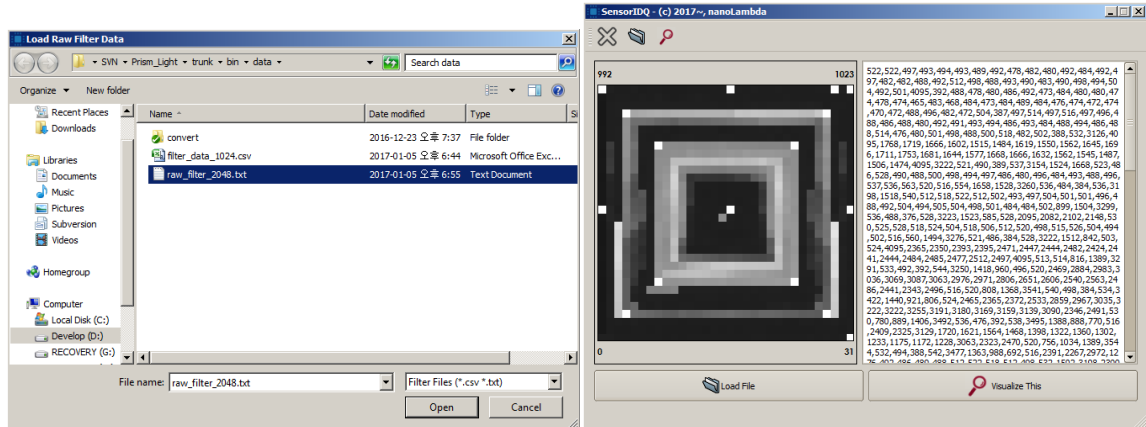
You can visualize (use  or  buttons) after copying the sensor data ('raw' or 'converted') you have acquired into the Text Window area of the GUI tool.

The following procedure shows how you can use SensorIDQ (Sensor Identity Query) tool to check the validity of raw sensor data or converted sensor data you have acquired. The usage of SensorIDQ tool is very simple, and the following figure shows the initial screen after SensorIDQ is executed. SensorIDQ GUI is divided into two main areas: 1) Filter Map (2D) area, 2) Text Window area. The Filter Map (2D) area shows your sensor data in a 2D matrix format (32×32 gray scale image format) that converts sensor data values in the range of 0 to 4095 to gray scale values in the range of 0 to 255 . Although the dynamic range of the sensor data is scaled down, it provides enough contrast to visually determine whether or not your sensor data is valid.

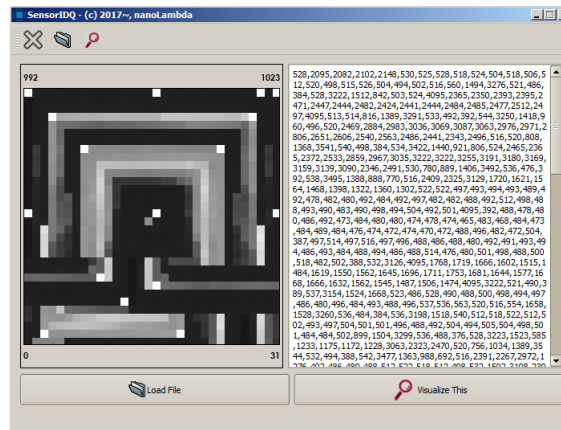


If you acquire sensor data from your own sensor interface board and save it as a comma separated text file (TXT or CSV format), you can visualize your data on the Filter Map (2D) in the following way .

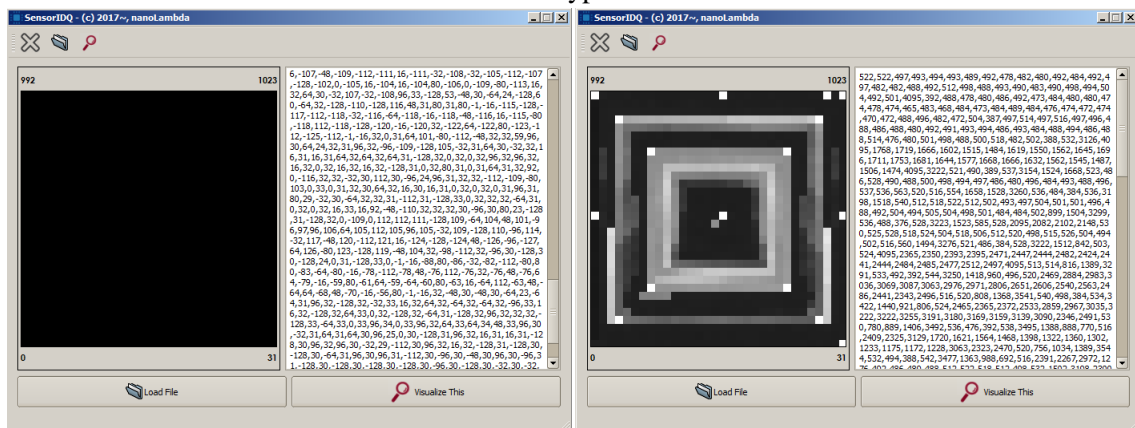
First select the sensor data file. You can store 2048 char array data, that is, data directly obtained from the sensor, in a text file. Or you can convert it to a 1024-size double type array, which is actually used in the spectra calculation and store it to file. SensorIDQ determines whether to pre-process (convert) or just visualize the data according to the number of data contained in your file. In other words, if the total number of data in file or Text Window is 2048, SensorIDQ judges this data as char type data, converts it into 1024 size double type, and finally displays it on 32×32 Filter Map (2D).



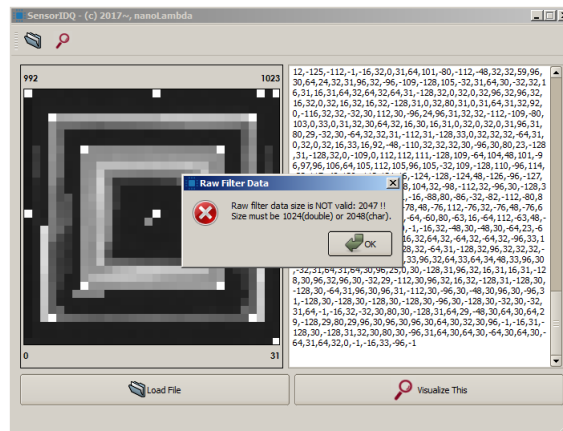
If your acquired 'raw' or 'converted' sensor data is corrupted, you can easily find the problem from the Filter Map (2D), as shown below. In this case, you must carefully trace back to your sensor interface board to find the root cause of this problem and to fix it.



Another way you can choose is "Copy & Paste" mentioned above as second way to sensor data validation. You can open your text file using a text editor, copy the contents of the file, paste it into the Text Window on the right side of SensorIDQ and press the "Visualize This" button. Whether or not visualization is performed after conversion is determined in the same manner as a method of loading-visualizing a text file. At this time, the transformed 1024 size double type data replaces the existing 2048 char array in Text Window. If necessary, you can copy 1024 size double type data in Text Window and use the data converted to double type.



If the sensor data you have acquired is corrupted, for example if the data size is not 1024 or 2048, the following warning message will be shown when attempting to display the sensor data on the FilterMap (2D). The following example shows a case where 1 byte in a 2048 char array tries to visualize the damaged data.



Appendix BUSB Driver Installation

Installing the USB Driver on Windows 7/8/10

When you first plug in your NSP32 spectral sensor, Windows will attempt (unsuccessfully) to automatically install the driver for your NSP32 spectral sensor. But, there will not exist any correct driver for your NSP32 spectral sensor in your Windows. So, you need to install USB driver by manual:

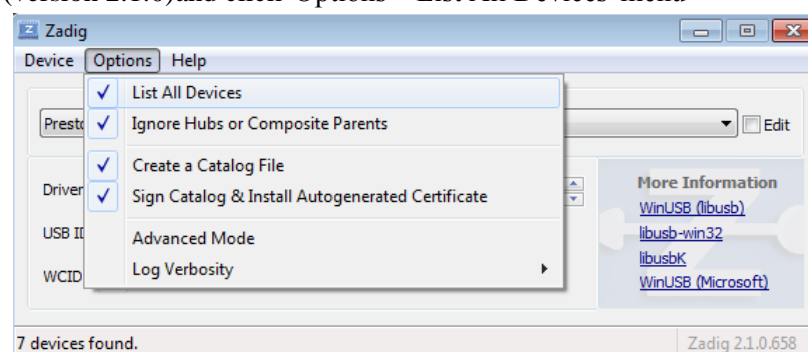
► Prerequisite

To install USB driver for your NSP32 spectral sensor, you have to have a Windows application (Zadig). You can get this program from '[NSP32_SDK_HOME]/tools/win32/zadig_2.1.0.exe' or download this freeware from a Web-site (<http://zadig.akeo.ie/>). You have to use version 2.1.0 of this application to install USB driver. Other versions will not install USB driver correctly.

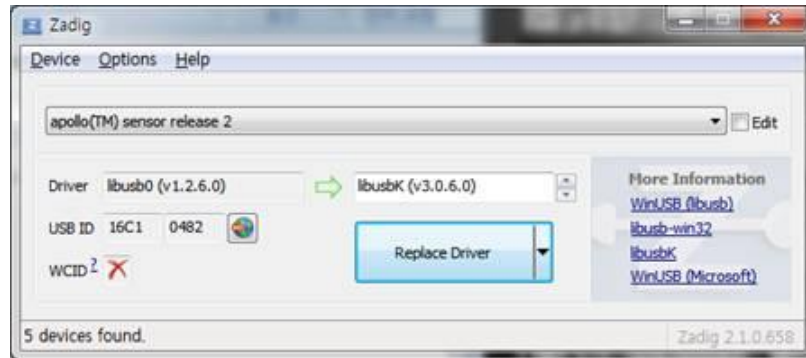
► Procedure

To manually install the new driver for your NSP32 spectral sensor:

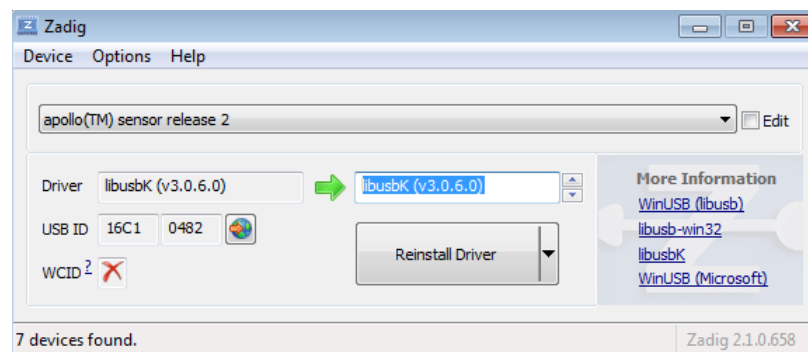
1. Make sure your NSP32 spectral sensor is plugged in.
2. Run 'Zadig' (version 2.1.0) and click 'Options→List All Devices' menu



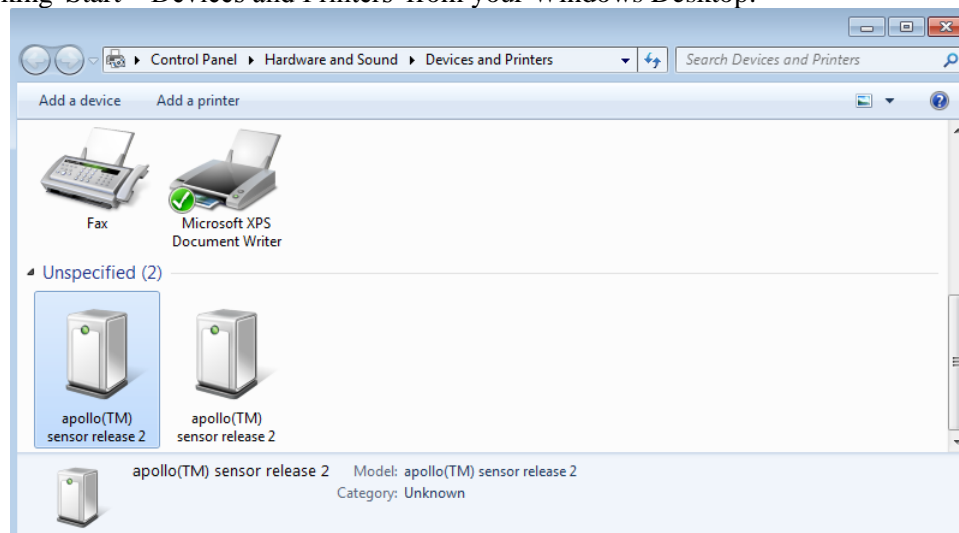
3. Select 'NSP32 spectral sensor' (or 'apollo(TM) sensor release 2'. This is old code name of NSP32 spectral sensor) from list box. If this is first time to install USB driver, you can see the current installed usb driver which is libusb0(v1.2.6.0). You need to select libusbK(v3.0.6.0) at right side and press 'Replace Driver' button.



4. If you already installed USB driver for NSP32 spectral sensor, you will see 'libusbK(v3.0.6.0)' is installed on your computer. In this case, you can re-install driver if you want to do it.



10. USB driver installation is done. You can check if driver is installed correctly or not by clicking 'Start→Devices and Printers' from your Windows Desktop:



Installing the USB Driver on Android, Linux, MacOS, and Raspbian

You don't need to install USB driver on Android, Linux, MacOS, and Raspbian platforms by yourself.

Appendix CNSP32 SDK Examples (High-Level API)

C/C++EXAMPLE: Calculate spectra with High-Level API

```
// examples sdk device spectrum.cpp : Defines the entry point for the console application.
// This console application is to show how programmer can create, use, and destroy of device
// object.
//
// (c) nanolambda 2012~2017

#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

#include"nsp_base_api.h"
#include"nsp_device_def.h"
#include"nsp_sdk_api.h"

/*
    Example: NSP32 device and spectrum APIs

    This example shows how user can use API to get raw filter data, spectrum data, and
    color info.
        1) acquire raw filter data from device
        2) reconstruct spectrum by spectrum API
*/
externchar* g_sensor_cal_file_path;

int examples_sdk_device_spectrum(bool dump_to_file)
{
    int    ret_value = 0;
    double *spec_output = NULL;
    double *wavelength_output = NULL;
    char str_sensor_id1[SENSOR_ID_STRING_LENGTH];
    char str_sensor_id2[SENSOR_ID_STRING_LENGTH];

    try{
        // initialize API for spectrum calculation
        ret_value = sdkCreate();

        if( ret_value > 0 ){
            // add one sensor to container
            int total_count_of_active_sensors
                = sdkRegister(g_sensor_cal_file_path);
            if( total_count_of_active_sensors == 0 ){
                throw"Fail to add one sensor to active sensor list.";
            }

            // get sensor ID from sensor calibration data.
            int id_size = sdkGetSensorIDFromCalData(str_sensor_id2);
            if( id_size == 0 ){
                printf("[examples_sdk_device_spectrum] Error: sensor ID is
                    not available in calibration data.\n");
            }

            // activate a physical device(sensor)
            int current_device_index = 0; // 1'st device (default)
            ret_value = sdkActivateSensorWithID(str_sensor_id2);
            if( ret_value == NSP_RETURN_VALUE_NO_MATCHED_SENSOR ){
                printf("[examples_sdk_device_spectrum] Error: there is no
                    matched sensor in your system: %s\n", str_sensor_id2);
                throw"there is no matched sensor in your system.";
            }
            elseif( ret_value == NSP_RETURN_VALUE_NO_MATCHED_CAL_DATA ){

```

```

        printf("[examples_sdk_device_spectrum] Error: there is no
        matched calibration data in your system: %s\n", str_sensor_id2);
        throw "there is no matched calibration data in your
            system.";
    }

    // get sensor ID from device (sensor).
    id_size = sdkGetSensorIDFromDevice(str_sensor_id1);
    if( id_size == 0 ){
        printf("[examples_sdk_device_spectrum] Error: sensor ID is
            not available in calibration data.\n");
    }

    // check if sensor ID is matched between device and sensor(device)
    // calibration data.

    if(strncmp(str_sensor_id1, str_sensor_id2, SENSOR_ID_STRING_LENGTH) == 0)
        printf("[examples_sdk_device_spectrum] Sensor ID is matched
            (%s <-> %s).\n", str_sensor_id1, str_sensor_id2);
    else {
        printf("*****\n");
        printf("[examples_sdk_device_spectrum] Error: Sensor ID is
            NOT matched (%s <-> %s).\n", str_sensor_id1, str_sensor_id2);
        printf("[examples_sdk_device_spectrum] Error: Spectrum (and
            color) data will not correct.\n");
        printf("[examples_sdk_device_spectrum] Solution: Get correct
            sensor calibration file for %s.\n", str_sensor_id1);
        printf("*****\n");
    }

    // OR, you can get optimal SS by AE
    int cur_ss = 10;
    if(1){
        cur_ss = sdkGetOptimalShutterSpeed();
        printf("SS by AE: %d\n", cur_ss);

        if( cur_ss == NSP_DEVICE_AE_SATURATION_INDICATOR ) //
        {
        }
        elseif (cur_ss == NSP_DEVICE_MAX_SS
            || cur_ss == NSP_DEVICE_MIN_SS )
        {
        }
    }
    else{
        cur_ss = 50;
        printf("SS by manual: %d\n", cur_ss);
    }
    // set shutter speed
    sdkSetShutterSpeed(cur_ss);

    // convert shutter speed to exposure time (ms) for your reference.
    int master_clock = NSP_DEVICE_MASTER_CLOCK;
    double exposure_time_val = 0;
    sdkShutterSpeedToExposureTime(master_clock, cur_ss,
        &exposure_time_val);
    printf("[examples_sdk_device_spectrum] current shutter speed: %d
        (exposure time = %.3f)\n", cur_ss, exposure_time_val);

    // calculate spectrum data
    int spectrum_data_size = sdkGetSpectrumLength();
    if( spectrum_data_size <= 0 ) throw "Invalid spectrum length.";
    printf("Spectrum data size: %d\n", spectrum_data_size);
    spec_output = (double *)malloc(spectrum_data_size*sizeof(double));
    if( spec_output == NULL )
        throw "Fail to allocate memory for spectrum data.";
    wavelength_output
        = (double *)malloc(spectrum_data_size*sizeof(double));
    if( wavelength_output == NULL )
        throw "Fail to allocate memory for wavelength data.";

    int frame_averages = 50;
    ret_value = sdkCalculateSpectrum(cur_ss, frame_averages,
        spec_output, wavelength_output);

```

saturated

```

        if( ret_value < 0 ){
            throw "Fail to calculate spectrum.";
        }

        // dump related information and data to CSV format file.
        if( dump_to_file == true ){
            // saving spectrumdata to csv
            std::string outputFileName =
                "examples_sdk_device_spectrum_color_"
                + std::string(str_sensor_id1) + "_result.csv";
            fstream filestream;
            filestream.open(outputFileName.c_str(), ofstream::out);
            if (filestream.is_open() == true) {
                filestream << "Sensor ID, "<<str_sensor_id1 << endl;
                filestream << "Shutter speed=", "<< cur_ss << endl;

                filestream << "Wavelength data"<< endl;
                for (int i = 0; i < (int) spectrum_data_size; i++) {
                    filestream << wavelength_output[i];
                    if (i != (spectrum_data_size - 1))
                        filestream << ",";
                }
                filestream << endl;
                filestream << "Spectrum data"<< endl;
                for (int i = 0; i < (int) spectrum_data_size; i++) {
                    filestream << spec_output[i];
                    if (i != (spectrum_data_size - 1))
                        filestream << ",";
                }
                filestream << endl;
                filestream.close();
            }
            printf("[examples_sdk_device_spectrum] Spectrum dump to
                \"%s\" -- done!\n", outputFileName.c_str());
        }

        if( spec_output )        delete spec_output;
        if( wavelength_output )  delete wavelength_output;

        // destroy SDK object.
        sdkDestroy();
    }
    return NSP_RETURN_VALUE_SUCCESS;
}
catch(nsp_base_exception &e){
    if( spec_output )        delete spec_output;
    if( wavelength_output )  delete wavelength_output;

    // destroy SDK object.
    sdkDestroy();

    printf("[examples_sdk_device_spectrum] NSP_BASE_EXCEPTION happened: %s\n",
        e.what());
    return -(int)e.error_code;
}
catch(...){
    if( spec_output )        delete spec_output;
    if( wavelength_output )  delete wavelength_output;

    printf("[examples_sdk_device_spectrum] std::exception happed !!\n");
    return NSP_RETURN_VALUE_FAILURE;
}
}

```

Appendix DNSP32 SDK Examples (Low-Level API)

C/C++ EXAMPLE: Calculate spectra with Low-Level API

```
// examples_device_spectrum_api.cpp : Defines the entry point for the console application.
// This console application is to show how programmer can create, use, and destroy of device
// object.
//
// (c) nanolambda 2012~2017

#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

#include"nsp_device_api.h"
#include"nsp_spectrum_api.h"
#include"nsp_sdk_common.h"

/*
    Example: NSP32 device and spectrum API

    This example shows how user can use API to get raw filter data and spectrum data.
    1) acquire raw filter data from device
    2) reconstruct spectrum by spectrum API
*/

int examples_device_spectrum_api(bool dump_to_file)
{
    int          current_device_index = 0;
    int          total_num_of_sensors = 0;
    // 50 frames are needed to have reliable/stable spectrum.
    int          frame_averages = 50;
    // user can change this for different illumination condition.
    int          new_shutter_speed = 150;
    double       sensor_data_buf[SENSOR_DATA_SIZE];    // raw sensor(filter) data
    double       background_data_buf[SENSOR_DATA_SIZE]; // background sensor data
    int          return_value = 0;

    try{
        char str_sensor_id[SENSOR_ID_STRING_LENGTH];
        char str_sensor_id2[SENSOR_ID_STRING_LENGTH];

        // open(connect) device (sensor)
        int total_num_of_sensors = duConnect();
        if( total_num_of_sensors <= 0 ){
            printf("[examples device spectrum api] Device is not connected.\n");
            return NSP_RETURN_VALUE_FAILURE;
        }

        // get sensor ID from physical device (sensor)
        duGetSensorID(str_sensor_id);

        // create Core Spectrum object for spectrum calculation
        if( csCreate() != NSP_RETURN_VALUE_SUCCESS ){
            throw"Fail to create Core Spectrum object.";
        }

        // add one sensor to active sensor list
        char sensor_cal_file_path[1024];
        sprintf(sensor_cal_file_path, "./config/sensor_%.s.dat", str_sensor_id);
        int total_count_of_active_sensors = csRegister(sensor_cal_file_path);
        if( total_count_of_active_sensors == 0 ){
            throw"Fail to add one sensor to active sensor list.";
        }
    }
}
```

```
// get sensor ID from sensor calibration data for matching.
int id_size = csGetSensorID(str_sensor_id2);
if( id_size == 0 ){
    printf("[examples_device_spectrum_api] Error: sensor ID is not
        available in calibration data.\n");
}

// check if sensor ID is matched between device and sensor(device)
calibration data.
if( strcmp(str_sensor_id, str_sensor_id2, SENSOR_ID_STRING_LENGTH) == 0 )
    printf("[examples_device_spectrum_api] Sensor ID is matched (%s <-
        > %s).\n", str_sensor_id, str_sensor_id2);
else {
    printf("*****\n");
    printf("[examples_device_spectrum_api] Error: Sensor ID is NOT
        matched (%s <-> %s).\n", str_sensor_id, str_sensor_id2);
    printf("[examples_device_spectrum_api] Error: Spectrum (and color)
        data will not correct.\n");
    printf("[examples_device_spectrum_api] Solution: Get correct sensor
        calibration file for %s.\n", str_sensor_id);
    printf("*****\n");
}

// get ADC settings info. from calibration data.
int    adc_gain = 0, adc_range = 132;
csGetSensorParameters(&adc_gain, &adc_range);

// change ADC settings of device
duSetSensorParameters(adc_gain, adc_range);

// another function to have total count of device
total_num_of_sensors = duGetTotalSensors();

for(int i=0; i<total_num_of_sensors; i++){
    int    current_device_index = i;

////////////////////////////////////
    // Acquire raw filter data

    // activate a specific device(sensor)
    return_value = duActivateSensorWithIndex(current_device_index);
    if( return_value == NSP_RETURN_VALUE_FAILURE ||
        return_value == NSP_RETURN_VALUE_UNKNOWN )
        return NSP_RETURN_VALUE_FAILURE;

    // get sensor id of currently activated device(sensor)
    return_value = duGetSensorID(str_sensor_id);
    if( return_value == NSP_RETURN_VALUE_SUCCESS )
        printf("[examples_device_spectrum_api] sensor index: %d,
            Sensor ID: %s\n", current_device_index, str_sensor_id);
    else
        memcpy(str_sensor_id, "unknown_ID", 10*sizeof(char));

    // get background data
    // set/change shutter speed of device(sensor)
    return_value = duSetShutterSpeed(1);

    // get one filter output (sensor data)
    return_value = duGetFilterData(background_data_buf, frame_averages);

    // set background data
    csSetBackground(background_data_buf);

    // get current shutter speed from device(sensor)
    // note: after sensor open (initialized), shutter speed is always 1.
    int old_shutter_speed = duGetShutterSpeed();

    // set/change shutter speed of device(sensor)
    if(1){
        // Get shutter speed with AE
        int    num_of_valid_filters = csGetNumOfValidFilters();
        int* valid_filters = csGetValidFilters();
        new_shutter_speed = duGetOptimalShutterSpeed(valid_filters,
```

```

        num_of_valid_filters);
    printf("[examples_device_spectrum_api] optimal SS by AE
           = %d\n", new_shutter_speed);
}

return_value = duSetShutterSpeed(new_shutter_speed);

// convert shutter speed to exposure time (ms) for your reference.
int     master_clock = NSP_DEVICE_MASTER_CLOCK;
double  exposure_time_val = 0;
return_value = duShutterSpeedToExposureTime(master_clock,
                                             new_shutter_speed, &exposure_time_val);
printf("[examples device spectrum api] current shutter speed: %d
       (exposure time = %.3f)\n", new_shutter_speed, exposure_time_val);

// get one filter output (sensor data)
return_value = duGetFilterData(sensor_data_buf, frame_averages);
printf("[examples_device_spectrum_api] sensor data at SS = %d\n",
       new_shutter_speed);
for(int j=0;j<SENSOR_DATA_SIZE;j++){
    if( j != 0 && j % 15 == 0 )        printf("\n");
    printf("%.2f, ", 7, sensor_data_buf[j]);
}
printf("\n");

////////////////////////////////////
// Calculate spectrum

double  filter_data_in[SENSOR_DATA_SIZE];
double  *spectrum_data_out = NULL;
double  *wavelength_data_out = NULL;
int      cur_ss = 10;

// use a sample data as input filter data.
memset(filter_data_in, 0, sizeof(double)*SENSOR_DATA_SIZE);
for(int i=0;i<SENSOR_DATA_SIZE;i++)
    filter_data_in[i] = sensor_data_buf[i];

// get spectrum data size from sensor calibration data
int      spectrum_data_size = csGetSpectrumLength();
printf("[examples_spectrum_core_api] spectrum data size: %d\n",
       spectrum_data_size);

// allocate memory for spectrum and wavelength data.
spectrum_data_out
    = (double *)malloc(sizeof(double)*spectrum_data_size);
if( spectrum_data_out == NULL )
    throw"fail to allocate memory for spectrum data.";
wavelength_data_out
    = (double *)malloc(sizeof(double)*spectrum_data_size);
if( wavelength_data_out == NULL )
    throw"fail to allocate memory for wavelength data.";

// get spectrum data
int ret_val = csCalculateSpectrum(filter_data_in, cur_ss,
                                spectrum_data_out, wavelength_data_out);
if( ret_val == spectrum_data_size ){
    for(int i=0;i<spectrum_data_size;i++){
        if( i != 0 && i % 3 == 0 )        printf("\n");
        printf("%4.0f(nm) - %.2f, ",
               wavelength_data_out[i], 7, spectrum_data_out[i]);
    }
    printf("\n");

    // dump related information and data to CSV format file.
    if( dump_to_file == true ){
        //////////////////////////////////////
        // saving spectrumdata to csv
        std::string outputFileName
            = "examples device spectrum api "
              + std::string(str_sensor_id)
              + "_result.csv";
        fstream filestream;
        filestream.open(outputFileName.c_str(),

```

```

ofstream::out);
if (filestream.is_open() == true) {
    filestream << "Sensor ID, "<<
        str sensor id << endl;

    filestream << "Raw filter data at SS="<<
        new shutter speed << endl;
    for (int i = 0; i < SENSOR_DATA_SIZE; i++) {
        filestream << sensor_data_buf[i];
        if (i != (SENSOR_DATA_SIZE - 1))
            filestream << ",";
    }
    filestream << endl;

    filestream << "Wavelength data"<< endl;
    for (int i = 0;
        i < (int) spectrum_data_size; i++) {
        filestream << wavelength_data_out[i];
        if (i != (spectrum_data_size - 1))
            filestream << ",";
    }
    filestream << endl;
    filestream << "Spectrum data"<< endl;
    for (int i = 0;
        i < (int) spectrum_data_size; i++) {
        filestream << spectrum_data_out[i];
        if (i != (spectrum_data_size - 1))
            filestream << ",";
    }
    filestream << endl;
    filestream.close();
}
printf("[examples device spectrum api] Spectrum dump
to \"%s\" -- done!\n", outputFile.c_str());
}
}

// Destroy Core Spectrum Object
csDestroy();

// close(disconnect) device(sensor)
duDisconnect();
return NSP_RETURN_VALUE_SUCCESS;
}
catch(nsp_base_exception &e){
    csDestroy();
    duDisconnect();

    printf("[examples device spectrum api] NSP BASE EXCEPTION happened: %s\n",
        e.what());

    return (int)-e.error_code;
}
catch(...){
    csDestroy();
    duDisconnect();

    printf("[examples device spectrum api] std::exception happed !!\n");
    return NSP_RETURN_VALUE_FAILURE;
}
}

```

Appendix ENSP32 SDK Examples (MATLAB Wrapper)

MATLAB EXAMPLE: Calculate spectra with MATLAB Wrapper

```
% spectrometer_example.m
%
% MATLAB example code for APIs
%
% Copyright 2016- nanoLambda, Inc.
% $Revision: 1.7.0.0 $ $Date: 2016/12/30 $
%

close all;clear all;clc;

LIBRARY_PATH = '\lib_win32\';

CORE_LIBRARY_NAME = 'CrystalCore.dll';
DEVICE_LIBRARY_NAME = 'CrystalPort.dll';

path(path,[cd , '/../wrappers/matlab']);
path(path,[cd , LIBRARY_PATH]);

% load API DLLs
[ret_device] = load_device_api_library([cd LIBRARY_PATH DEVICE_LIBRARY_NAME]);
[ret_core] = load_core_api_library([cd LIBRARY_PATH CORE_LIBRARY_NAME]);

if( ret_core ~= 1 || ret_device ~= 1 )
    disp(['Error] Fail to load API libraries.']);
return;
end

% initialize Device object.
[ret] = connect_device();
if( ret < 0 )
    disp(['Error] No sensor in your system.']);
return;
end

% get sensor ID of physical spectral sensor
[ret,SENSOR_ID, total_count] = get_sensor_ID_from_device();
disp(['sensor ID from device: ' SENSOR_ID]);

%%
% create Core Spectrum object.
[ret] = create_core_spectrum_object();

% add one calibration file path to Core Spectrum object
sensor_cal_file_path = [cd 'sensor_' SENSOR_ID '.dat'];
[ret] = add_to_sensor_data_list(sensor_cal_file_path);
if ret == -1
    disp('Fail to initialize API.'];
return;
end

% get spectral sensor ID from calibration data
[ret, SENSOR_ID2, total_count2] = get_sensor_ID_from_cal_data();
disp(['sensor ID from cal data: ' SENSOR_ID2]);

% get sensor parameters (ADC gain and range) from calibration data
[ret,adc_gain, adc_range] = get_sensor_parameters_from_cal_data();

% get wavelength info.
[ret,START_WAVELENGTH, END_WAVELENGTH, WAVELENGTH_STEP] = get_wavelength_info_from_cal_data();
WAVELENGTH_RANGE = (START_WAVELENGTH:1:END_WAVELENGTH);
WAVELENGTH_STEP = 1;
```



```
% activate a specific sensor with sensor ID
activate_sensor_with_ID(SENSOR_ID2);

% set sensor parameters to physical spectral sensor.
[ret] = set_sensor_parameters_to_device(adc_gain, adc_range);

FRAME_AVERAGE = 10;

% get background data at ss=1
set_shutter_speed_to_device(1);
[ret,background_data] = acquire_sensor_data(FRAME_AVERAGE);

% set background to Core Spectrum object.
[ret] = set_background_data(background_data);

SHUTTER_SPEED = 50;
set_shutter_speed_to_device(SHUTTER_SPEED);

% set acquisition conditions to acquire spectrum data.
if ret ~= -1
    figure;
    for frame_i=1:10
        % acquire raw sensor data from spectral sensor
        [ret,filter_data] = acquire_sensor_data(FRAME_AVERAGE);

        if( isempty(filter_data) == 0 )

            % calculate spectrum by Core Spectrum API
            [ret,spec_data, wavelength_data] = calculate_spectrum(filter_data, SHUTTER_SPEED);

            % just check
            if length(spec_data) == 0
                continue;
            end

            % calculate color with spectrum and wavelength data
            [ret,X, Y, Z, R, G, B, x, y, z, cct] = calculate_color(spec_data, wavelength_data);

            % plot it

            subplot(1,2,1);
            plot(filter_data);
            xlabel('Filter Index');
            ylabel('Power (Relative)');
            title('Raw Sensor Data Graph');
            axis([-Inf Inf 0 4096]);
            grid on;

            subplot(1,2,2);
            plot(wavelength_data, spec_data);
            xlabel('Wavelength (nm)');
            ylabel('Power (Relative)');
            title(sprintf('Spectrum Graph\nXYZ=(%.3f,%.3f,%.3f)\nRGB=(%d,%d,%d),CCT=%.3f,X,Y,Z,R,G,B,cct));
            axis([-Inf Inf 0 Inf]);
            grid on;
            drawnow;

        end
        disp(sprintf('%d-th run for spectrum calculation.', frame_i));
    end
end

% disconnect device API
[ret] = disconnect_device();

% destroy APIs (core spectrum and color application)
[ret] = destroy_core_spectrum_object();

% unload DLLs
%% 2017.01.04 - currently, 'unloadlibrary()' function with 'CrystalPort.dll'
% have one problem (crash). So, before nanoLambda fix this problem, please
% don't try to unload 'CrystalPort' library explicitly.
[ret] = unload_api_library('CrystalPort');

[ret] = unload_api_library('CrystalCore');
```

Appendix FNSP32 SDK Examples (Python Wrapper)

Python EXAMPLE: Calculate spectra with Python Wrapper

```
import sys
import csv
import ctypes
sys.path.append(".././../wrappers/python")
if(sys.version_info[0] < 3):
    from wrapper_python2 import *
    from wrapper_python2.core import *
    from wrapper_python2.device import *
    from wrapper_python2.color import *
    print("*****")
    print("[Python-2]      Python Version : ", sys.version_info.major, ".", sys.version_info.minor, " Detected")
    print("*****")
else:
    from wrapper_python3 import *
    from wrapper_python3.core import *
    from wrapper_python3.device import *
    from wrapper_python3.color import *
    print("*****")
    print("[Python-3]      Python Version : ", sys.version_info.major, ".", sys.version_info.minor, " Detected")
    print("*****")

#Initialization
if sys.platform == 'win32':
    initialize("../Libs/CrystalBase.dll")
    pSpecCore = initialize_core_api("../Libs/CrystalCore.dll")
    pSpecDevice = initialize_device_api("../Libs/CrystalPort.dll")
else:
    initialize("../Libs/libCrystalBaseLight.so")
    pSpecCore = initialize_core_api("../Libs/libCrystalCoreLight.so")
    pSpecDevice = initialize_device_api("../Libs/libCrystalPortLight.so")

initialize_color_api(pSpecCore)

connectReturn = connect_device(pSpecDevice) # return total num of devices connected with system

if connectReturn > 0:

    (ret, sensorID) = get_sensor_id_device(pSpecDevice)

    create_core_object(pSpecCore)

    if sys.platform == 'win32':
        csInit_Return = load_sensor_file(pSpecCore, b"..\\config\\sensor_" + sensorID + b".dat")
    else:
        csInit_Return = load_sensor_file(pSpecCore, b"..\\config\\sensor_" + sensorID + b".dat")

    (ret, sensorID) = get_sensor_id_file(pSpecCore)

    get_sensor_parameters_from_device(pSpecDevice)

    (adcGain, adcRange) = get_sensor_parameters_from_calibration_file(pSpecCore)

    settingReturn = set_sensor_parameters_to_device(pSpecDevice, adcGain, adcRange)

    total_num_of_sensors = total_sensors_connected(pSpecDevice)

    get_capacity_sensor_data_list(pSpecCore)

    for index in range(total_num_of_sensors):
```

```

#activate a specific device(sensor)
activatingReturn = index_activation(pSpecDevice,index)

#get sensor id of currently activated device(sensor)
(ret, sensorID) = get_sensor_id_device(pSpecDevice)

#get and set shutter speed of device(sensor)
get_shutter_speed(pSpecDevice)
set_shutter_speed(pSpecDevice,1)

#get one filter output (sensor data)
filterData = get_filter_data(pSpecDevice,20)

#set background data
set_background_data(pSpecCore,filterData)

#get and set shutter speed of device(sensor)
get_shutter_speed(pSpecDevice)

#Get shutter speed with AE
newSS = get_optimal_shutter_speed(pSpecDevice)
set_shutter_speed(pSpecDevice,newSS)

#convert shutter speed to exposure time (ms) for your reference
ss_to_exposure_time(pSpecDevice,5,newSS)

filterData = get_filter_data(pSpecDevice,20)

specSize = get_spectrum_length(pSpecCore)
(ret, specData,wavelengthdata) = calculate_spectrum(pSpecCore,filterData,newSS)

(Start_Wavelength, End_Wavelength, Interval_Wavelength) = get_wavelength_information(pSpecCore)

get_resolution(pSpecCore)

colorData = calculate_color_data(pSpecCore,specData, wavelengthdata,specSize)

if sys.version_info[0] < 3:
    fileName = (r"SpecrtumData2_" + sensorID + ".csv");
    data = []
    for i in range(get_spectrum_length(pSpecCore)):
        data.append(str(specData[i]).split(","))

    with open(fileName, "wb") as csv_file:
        writer = csv.writer(csv_file, delimiter=',')
        for line in data:
            writer.writerow(line)
    csv_file.close()
else:
    fileName = (b"SpecrtumData3_" + sensorID + b".csv");
    data = []
    for i in range(get_spectrum_length(pSpecCore)):
        data.append(str(specData[i]).split(","))

    with open(fileName, 'w', newline='') as csvfile:
        filewriter = csv.writer(csvfile, delimiter=',',
                                quotechar='|', quoting=csv.QUOTE_MINIMAL)
        for line in data:
            filewriter.writerow(line)
    csvfile.close()

else:
    print ("*****")
    print ("[PrismError]Device Not Connected. Please connect Device and try again.")
    print ("*****")
close_color_api(pSpecCore)
close_core_object(pSpecCore)
disconnect_device(pSpecDevice)

```