PS Software Development Kit.

Programming Reference

1. Table of Contents

2. Introduction

PS-SDK is a Software Development Kit for developers, which is intended for controlling Canon[1] PowerShot[1] cameras from a PC.

Using PS-SDK allows developers to implement following features in software:

- To manage a list of cameras connected to the PC;
- To establish connection with a camera and to close it;
- To manage camera settings and image shooting properties;
- To receive live-view image from camera viewfinder;
- To take photos;
- To download image files from the camera.

Basic topics

PS-SDK provides a C language interface for accessing digital cameras and data created by these cameras. PS-SDK is designed to provide standard methods of accessing different camera models and their data.

Modern Canon digital cameras do not provides capabilities for controlling cameras from the PC. By default only access to camera's flash-card is possible.
To enable camera control features it is required to run additional software on the camera which should handle controlling commands and perform corresponding actions directly in the camera. This task is performed by PS module component of the PS-SDK. When started, it switches the camera to PC-controlled mode and the camera becomes ready to process commands from the PC. After finishing of the PS module, the camera returns to standard work mode.

System requirements

- OS: Microsoft Windows 8.x/7/Vista/XP
- OS Architecture: 32/64 bit
- RAM: 1Gb
- Hard disk space: 50 megabytes
- Camera connection interface: USB 2.0
- SD/SDHC Flash memory card for each camera
- SD/SDHC Card Reader

Supported cameras

1. Canon PowerShot SX400 IS Revision 100B
2. Canon IXUS 145 / PowerShot ELPH 135 Revision 100C
3. Canon PowerShot G15 Revision 100B/100E
4. Canon PowerShot SX510 HS Revision 100C
5. Canon PowerShot SX170 IS Revision 100A/101A
6. Canon PowerShot A1400 Revision 100B
7. Canon PowerShot SX500 IS Revision 100C/100D
8. Canon PowerShot SX160 IS, revision 100A
9. Canon PowerShot A4000 IS, revisions 100C/101A/101B/102A
10. Canon PowerShot A810, revisions 100B/100D/100E

[1] Canon and PowerShot are registered trademarks of the Canon Inc.

11. Canon PowerShot SX40 HS, revisions 100G/100I
12. Canon PowerShot A2200, revision 100D
13. Canon PowerShot SX150 IS, revision 100A
14. Canon PowerShot A800, revisions 100B/100C
15. Canon PowerShot A800, revision 100B
16. Canon PowerShot SX130 IS, revisions 101C /101D/101F
17. Canon PowerShot A495, revisions 100E/100F

SDK Directory structure

PS-SDK distribution contains following directories structure:

Doc – SDK documentation;
PSSDK C SDK files:
inc header files directory contains **pssdk.h** and **psproperty.h** files;
lib pssdk.lib file
redist PSSDK.dll file for redistributing SDK with client applications
PSSDK.NET .NET SDK file PSSDK-NET.dll;
Samples source code sample directories:
VC# Sample C# samples for Visual Studio 2012;
VC++ Sample C samples;
Samples bin compiled binary examples of SDK using
VC# Sample bin binary samples written in C#;
VC++ Sample bin binary samples written in C;
Tools SDK auxiliary tools:
CardSetup – **CardSetup** utility for installation PS module on SD cards;
PSDriver – special device driver installation utility.
Activation – activation utility for licensing based on installation
CameraInfo – utility to get camera serial number for licensing based on camera
AutoShoot – utility for general testing proposes, automatically shoots with interval

Preparation to develop programs using the SDK

To use the PS-SDK using following header files are required:
pssdk.h , ***psproperty.h***

Be sure to copy the two header files listed above into the header access folder of the development environment. Be sure to add them to the application project workspace.
After header files are included, it is necessary to link the PS-SDK library.

There are two methods of linking PS-SDK: one where ***PSSDK.lib*** files are copied to the folder specified by a development environment library path and PSSDK.lib is specified as an import module, and another where ***PSSDK.dll*** is loaded by the **LoadLibrary** function.
When loading ***PSSDK.dll***, get pointers to each PS-SDK function using the **GetProcAddress** function and assign them to function pointer variables. When calling each PS-SDK function, make the call via the function pointer variable obtained here.

Redistribution of the PS SDK components on the client's PC

Before PS-SDK could be used, following steps should be performed:

1. Camera preparation: To enable camera-control feature, PS module should be written on camera's flash-card. This task is performed by card preparation utility CardSetup.exe, for each controlled camera;
2. Driver installation to the PC. Installation is performed only once, by running corresponding executable file;
3. Specified dll file should be placed into the system;

Installation files:

- ***CardSetup.exe***
- ***ps-sdk-driver-setup.exe***
- ***PSSDK_32.DLL***
- ***PSSDK_64.DLL***

USB driver installation

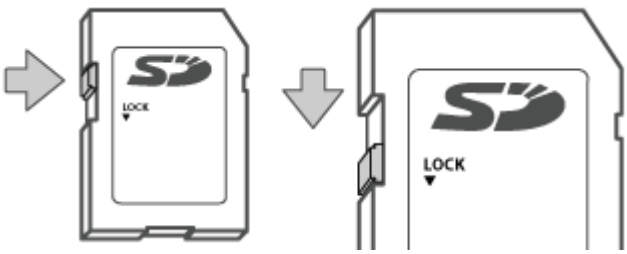
WinUSB is a universal driver for USB devices in the Windows system. The driver provides direct interaction with a device. WinUSB is supported by all current versions of the Windows system, beginning from the Windows XP. To automate driver installation procedure, utility included in the SDK, could be used. The utility builds driver and installs it to the system.
It is recommended to keep camera switched off while the driver is being installed. Switched on camera in itself is not an obstacle for driver installation, but driver couldn't be installed if Windows drivers installation wizard is opened. If camera in PC-controllable mode was connected to the PC before driver installation, the system would open Windows drivers installation wizard and to install driver, the wizard should be closed first.
Driver should be installed only once per system. Driver's installation does not affect camera's functioning in standard mode (when camera works without installed PS module).
Distributions of programs based on PS-SDK should include driver's installation archive.

Camera preparation

To response on PS-SDK commands, camera should run compatible PS-module. For each camera model and either for each camera firmware version (a camera could be produced with different firmware versions) suits different PS-modules. PS-module is installed to a flash-memory card compatible with the camera;

To install PS-module to a camera following steps should be performed:

1. PS-module version compatible with the camera selected;
2. PS-module, specific directories and marks structure are written to camera-compatible card;
3. The card is switched to the "lock" mode and is inserted to the camera. The "lock" mode indicates camera that the card contains additional module which should be run upon the camera start.



Switching camera memory card to "lock"state

There is a **CardSetup.exe** utility provided with the SDK. The utility simplifies installation of the PS-module. The utility recognizes exact camera model and its firmware version by photo-image

made by the camera. Then, it prepares the card and writes corresponding PS-module version to it. After that all that remains is to switch the card to “lock” mode and to insert the card to the camera.

When the camera, with the card in “lock” mode inserted, is switched on, PS-module would start and switch the camera to PC-controllable mode and the camera would become ready to perform commands received from a PC.
To return the camera to conventional work mode it takes you only to pull the card out of the camera or to switch the card to “unlock” mode. Consequent insertion of the card to the camera, or switching the card to “lock” would switch the camera to the PC-controllable mode again.

There are several possible ways of PS-module distribution along with solutions based on PS-SDK. E.g., you can prepare and deliver cards compatible with user cameras, if you have access to ones. Or, you might as well distribute utility itself to your users, in other case.

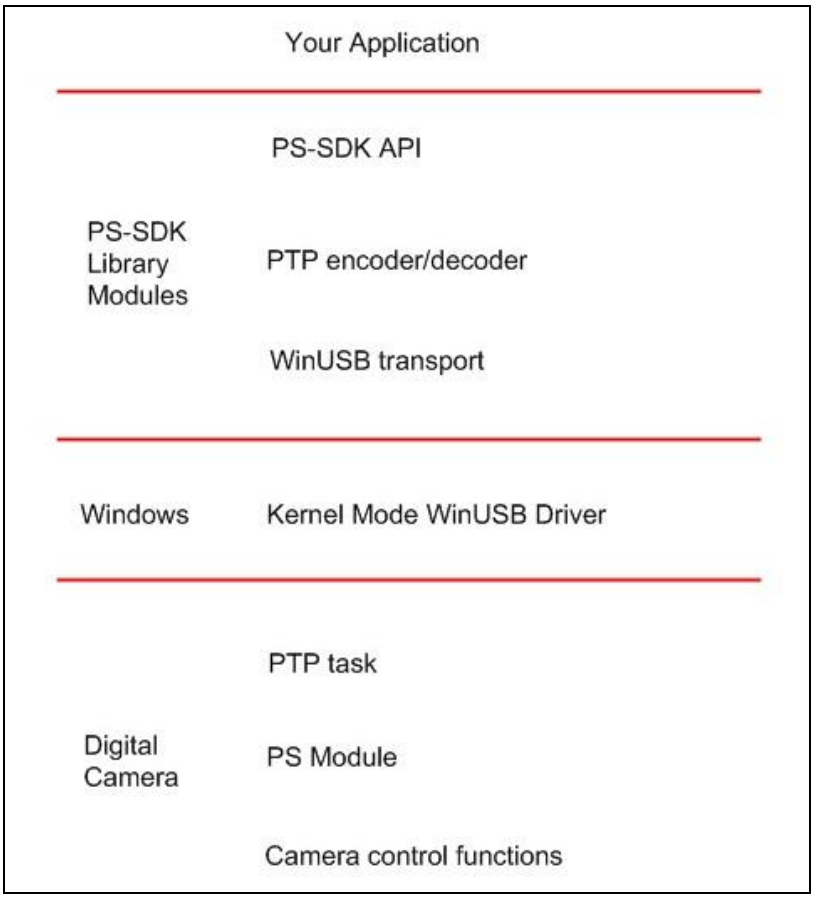
3. Overview

Protocol for Remote Connection

Commands are transferred from PC to a camera via camera USB connection using PTP protocol. PTP is an abbreviation of “Picture Transfer Protocol”

System architecture

Communication with a camera in the Windows system is provided by WinUSB driver. The driver creates separate device entries for camera in pc-controlled mode and for the same camera in standard mode, therefore, from the system’s point of view, the camera in pc-controlled mode and the camera in standard mode are two independent devices.
To provide full-scale communication with a camera specific PTP protocol extension is added to it via PS module installation (see p. 0)
Such feature prevents any possible conflicts with software which use the camera in standard mode (when camera works without installed PS module).



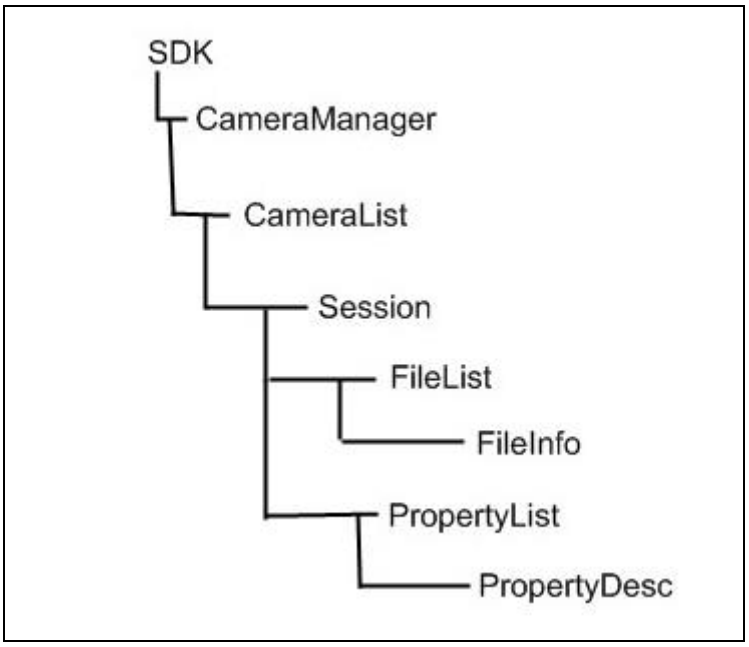
PS-SDK architecture

Typical interaction with the camera consists of following steps:

1. The SDK provides communication with WinUSB driver, generates PTP-command and sends it to the camera;
2. PS-module in the camera receives command, executes it and, if necessary, send data back to the PC;
3. The SDK receives data and decodes it.

Additionally, the SDK monitors camera's internal events and reports them to client-side software

PS-SDK objects



PS-SDK objects structure

PS-SDK employs a hierarchical structure with a CameraManager object at the root in order to control and access cameras connected to the host PC. This hierarchical structure consists of the following elements:

- **CameraManager** – an enumeration of all compatible cameras connected to the system via USB interface.
- **CameraInfo** – structure which describes given camera
- **Session** – camera controlling object
- **FileList** – collection with descriptions of files captured during the current session;
- **FileInfo** – structure which describes given captured file
- **Prop_Desc** – structure which describes given property. Description includes possible values and various auxiliary information;

Properties

Properties are stored under PS-SDK for camera and image objects. For example, properties may represent values such as camera **Av** and **Tv**. The functions **PSGetPropertyData** and **PSSetPropertyData** are used to get and set these properties. Since this API takes objects of undefined type as arguments, the properties that can be retrieved or set differ depending on the given object. In addition, each property has a list of currently settable values. **PSGetPropertyList** is used to get this list of settable values. **PSGetPropertyDesc** returns description of the given property.

For details on types of properties, the objects they are associated with, and the role they play, see p. 0.

Following properties are represented in the SDK:

Property name	Description
PS_ShootingMode	shooting mode (read-only)
PS_ImageSize	image size
PS_JpegQuality	image quality
PS_MeteringMode	metering mode

PS_WBmode	white balance mode
PS_FlashMode	flash mode
PS_RedEyeMode	red-eye mode
PS_ISOSpeed	ISO
PS_Av	Aperture
PS_Tv	shutter speed
PS_ExposureComp	exposure compensation
PS_FlashComp	flash compensation
PS_AFMode	autofocus mode
PS_BatteryLevel	battery level (read-only)
PS_FocusingZone	focusing zone
PS_ManualFocusMode	switching manual focus mode
PS_DirectTransfer	switching direct transfer mode

Events

Asynchronous events is a mechanism used to issue notifications from the SDK to the application regarding cameras connected to the host PC or state changes that have occurred for a camera. For example, if a state change occurs where a camera's shooting mode changes and a new image that needs to be transferred to the PC has been shot, a notification of that fact is sent to the application regardless of its state (asynchronously).

An event handler capable of the specific processing required for a particular event must be registered in order to receive such an event (notification). An event handler is a user function called when an event is received. Event handlers are also referred to as "callback functions." Users can allow events to be accepted by creating and registering callback functions that accept events issued by PS-SDK.
When an event occurs, the PS-SDK executes the callback function registered by the user. In dependence of event type, callback function could receive different auxiliary information about event. All callback functions receive so-called context information defined during event subscription
The user must release event handlers as they become unneeded using **PSFree** command.

Event types provided by PS-SDK represented in the table below. Table columns contain event names, short descriptions and event subscription function names. Subscription and callback functions described in details in corresponding sections of API reference (see p. 4)

Event name	Short description	Subscription function
PSCameraInitializationError	Event is activated when procedure of establishing connection session with a camera by means of **PSOpenSession** function is failed by some reasons	**PSCameraInitializationErrorSubscribe**
PSCameraListChanged	Event is activated when new compatible camera connected to the PC or previously connected camera is disconnected from the PC	**PSCameraListChangedSubscribe**
PSNewPreviewFrame	Event is activated when preview image is ready to be downloaded from the connected camera	**PSNewPreviewFrameSubscribe**

PSZoomChanged	Event is activated when a user changes zoom state of the camera	**PSZoomChangedSubscribe**
PSZoomCompleted	Event is activated when Zoom changing action, started by **PSSetZoom** command, is successfully finished	**PSZoomCompletedSubscribe**
PSShootCompleted	Event is activated when photo shooting process, started by **PSShoot** command, is completed	**PSShootCompletedSubscribe**
PSDownloadCompleted	Event is activated when image file downloading process, started by **PSDownload** or **PSDownloadTo** command, is completed	**PSDownloadCompletedSubscribe**
PSPropertyListChanged	Event is activated when a property becomes available or unavailable due to changing of other property value or switching camera to different mode	**PSPropertyListChangedSubscribe**
PSPropertyChanged	Event is activated when value of one of the properties is changed.	**PSPropertyChangedSubscribe**
PSFocusChanged	Event is activated only in manual focus mode when value of focus value changed.	**PSFocusChangedSubscribe**

Controlling a camera using the SDK

SDK Initialization and termination

PSInitializeSDK initializes SDK, **PSTerminateSDK** finishes its work and releases all allocated resources

Accessing camera

To access a camera following commands should be performed:

PSGetCameraManager command which initializes camera manager;
PSGetCameraList command returns collection of all compatible cameras connected to the PC;

To open camera connection session, call **PSOpenSession** command for camera item in cameras collection. With each camera only one connection could be established at one moment. It is possible to establish several simultaneous sessions with different cameras;
When work is finished, **PSFree** command should be called for all **PSSessionHandle** and **PSCameraManagerHandle** objects.

PSCameraListChangedSubscribe command makes subscription for camera list change event. The event's callback function receives **PSCameraConnectionState** value which describes change type: connection of new camera or disconnection of previously connected camera.

PSGetCameraConnectionState command checks whether camera connected to the PC in the session or not, using session's **PSSessionHandle**. Such feature permits you to easily process camera list connections events and to check connection status for camera with opened session (current camera).

Work with preview

The SDK permits programmer to receive image from camera's viewfinder. To enable receiving of preview frames to the PC call **PSSetPreviewState** command with **PS_PREVIEW_ENABLED** value as argument. Note that the camera always creates preview when it is in shooting mode. **PSSetPreviewState** only controls transferring the preview to the PC.

> **Important!** Enabling of receiving preview images to the PC significantly increases system's load.

The SDK receives preview as frequent as it possible. When another image is received, **NewPreviewFrame** event is activated. After receiving the event, image file could be read by **PSGetPreviewFrame** function and shown on the screen.
Preview size is defined by the camera parameters. Preview image size is at least 320x 240, and could be greater in new camera models. Preview images are received in BMP format.
To stop preview capturing mode, call **PSSetPreviewState** command with **PS_PREVIEW_DISABLED** value as argument. Attempt of reading current frame after stopping preview handling, raises an error.

Taking photos

To start a process of taking photo, call **PSShoot**. The process consists of following steps:

Camera flash is charged, if necessary. Charging process could take some time, so it may lead to pause before shooting
Auto-focus and auto-exposure are performed;
Photo is exposured
Shutter operates
Captured data is processed in the camera
Image file is written to the flash-card by default, or in case of using DirectTransfer mode image is transferred to PC memory.

When shooting process is over, **PSShootCompletedCallback** event is received. After shooting finishing, image file still in the camera card and is not sent to the PC yet. When photo-shooting routines are being processed, all other commands to the camera are blocked and preview is not available.

Transferring of captured photos

PSShootCompleted event receives **PSFileinfo** structure which contains information of photo taken, including identifier for its downloading.
All photos taken during current connection session are gathered in **PSGetFileList**.collection. To download photo-image file, call **PSDownloadFile** for necessary file. File would be downloaded to temporary folder and your application would receive full path to the file. **PSDownloadFileTo** command save file to the path defined by parameters.
PSDownloadCompletedCallback event is activated when download is completed. One of the event arguments – absolute path to the downloaded file.
When photo is already downloaded to the PC, this file on the memory card should be removed by **PSDeleteFile** command, file would be deleted from the card physically and its restoration wouldn't be possible. Image files should be removed from the card to free some place on it.
In case of using DirectTransfer mode same actions must be performed: download and delete files to avoid inappropriate memory usage.

Optical zoom operations

Camera's optical zoom is controlled by **PSSetZoom** command. The command receives current zoom value which could be in range from 0 to maximum zoom value, stored in **PSGetMaximumZoom** variable.
When zoom value is being set by the camera, all other commands are blocked, except command related with the preview mode. When zoom is finally set, **ZoomChanged** event is activated. **PSGetZoom** command returns current zoom value.
Zoom changing is also possible directly from the camera. When zoom is being changed from the camera, its value changes are immediately reported by the **ZoomChanged** event.

Work with properties

Camera parameters management is performed by its properties. **PSGetPropertyList** function returns list of all properties accessible at the moment (depends from camera model and its work mode) Description for given property could be received by **PSGetPropertyDesc** command. Description contains list of available values for property (depends from camera work mode and values of other properties) and property's type (read-only or read-write).

PSGetPropertyData returns current value of the property.
PSSetPropertyData command defines new values for read-write properties.

When list of available properties, or list of available values for any property, is changed, **PropertyListChanged** event is activated. On property's value changing, **PSPropertyChangedCallback** is activated**.** Property's value could change when other property's value is changed, camera is switched to different mode, battery's level decreased or increased, and so on.

Readable name of property is returned by **PSGetPropertyName** function
Readable name of property's value is returned by **PSGetPropertyValName** function.

Manual focus mode

For some camera models manual focus mode is supported. You need to switch **PS_ManualFocusMode** to On value prior of using other manual focus mode functions.

Check current focus value with **PSGetFocus** function. Move focus to desired distance value with **PSSetFocus** function. Distance value is the focusing distance in millimeters. There is special value '-1' for infinity focus. Please, note, camera have exact supported focus values, you can't use any distance value, but camera will move to closest supported value. Values are different for every zoom step of the camera. When using **PSSetFocus** function you need to use **PSFocusChanged** event for checking real focus value with **PSGetFocus** right after moving manual focus and use returned value as exact focus value.

As camera will fix real focus value every time you move manual focus it is impossible to use manual control for focus tuning with **PSSetFocus** functions. Use **PSFocusShift** function in this case. It uses exact supported values for manual focus and you can move focus value step by step with this function. You can move focus with both directions using positive and negative values and number of steps. Focus shift limited by minimal and maximum focus values, you can't move out of this limits.

DirectTransfer mode

By-default captured images saved on camera SD card prior to downloading to PC. For some camera models DirectTransfer mode available. When DirectTransfer mode is used captured image do not

saved on SD card in camera but transfers to PC directly. It is not possible to turn DirectTransfer mode on and off during camera work. You have to set SDK operation mode in using DirectTransfer mode if needed. Use \Tools\DirectTransfer\PSSDK.cfg file to turn DirectTransfer mode on.
Place .cfg file to SDK folder. Following value turns DirectTransfer mode on:

<UseDirectTransferIfAvailable val="true"/>

When DirectTransfer turned on SDK will check if mode is available on connected camera and turns it on. For camera without DirectTransfer mode support SDK will operate with saving images on SD card.
Direct transfer mode available for following camera models: Canon PowerShot SX170 IS, Canon PowerShot SX160 IS.

Additional features

All objects created during using the SDK require to free memory after work with them is finished. For this task **PsFree** command is used.

After **PSCameraErrorCallback** event activation for a session, any actions with the session would raise a error.

If camera initialization ends with error, **PSCameraInitializationErrorCallback** event is activated for **CameraManager** object. There are number of possible error reasons: low battery charge, unsupported camera mode etc.

If camera's battery is nearly empty and camera couldn't work adequately, **PSCameraErrorCallback** event is activated with **PS_LOW_BATTERY_LEVEL** argument.

It is recommended to supply controlled cameras with constant power sources.
PSProp_BatteryLevel property, which shows current charge level, also indicates supply from constant power source by its l value **PS_BatteryLevel_DC**.

All SDK operations are being logged. Log stores all information on SDK work and performed commands. Log-file is being written to Windows user local application data folder. By default:
Windows Vista/7/8

C:\Users\[User Account Name]\AppData\Local\psdll.log

Windows XP

C:\Documents and Settings\[User Account Name]\Local Settings\Application Data\psdll.log

Log files limited by 1Mb file size, older strings above that size will be removed at next SDK start. There are two PS-SDK log files:
psdll_api.log – stores all PS-SDK API calls to control using of SDK from application
psdll.log – stores internal infromation about PS-SDK operations

PS-SDK Basic types

This section introduces the basic data types used under the PSSDK. These data types are defined as C language types.

```
typedef int    CameraId;
typedef int    FileId;
```

PS-SDK Errors

Most of the APIs supplied by PS-SDK return an error code of type **PSResult** as their return value.
The return value of an API that terminates normally is **PS_OK**. If an error occurs, the return value of the API in question is set to the error code indicating the root cause of the error and any

passed parameters are stored as undefined values. (Note that an API used to control files is not limited to returning an error related to file control.)

For error codes, see the list given in the header file **PSSDK.h** or see PS Errors list in p. 0

4. API Reference

API Details

API specifications are explained in the following format.

Description:
Indicates the main API function

Syntax:
PSResult PSXXXXX(XXX inXXXX, XXX *outXXX) ;
Indicates the syntax for calling the API function.

Parameters:
Explains each argument in the syntax individually.

In the syntax, argument names in the format **inXXX** represent arguments for which you enter values. Argument names in the format **outXXX** represent arguments with values set by the libraries (that is, passed by reference). Before calling APIs, you must prepare variables for storing the data to be retrieved.

Return Values:
Explains API return values

See Also:
Indicates information related to the API.

Note:
Considerations when using the API.

Example:
Sample code.

PSInitializeSDK

Description:
Initialization of the SDK. Should be called before using SDK functions. Before initialization PS-SDK functions behavior is not defined.

Syntax:
PSResult PSSDKAPI PSInitializeSDK()

Parameters:
None

Return Values:
Returns **PS_OK** if successful, for other values see PS errors list, p. 0

See Also:
PSTerminateSDK

Note:
There is only one instance of the SDK could run in the system at the moment. If

SDK instance is already started, initialization would return specific error.

Example:

See code example in p. 0

PSTerminateSDK

Description:

Closes SDK and releases all resources allocated by the libraries

Syntax:

void PSSDKAPI PSTerminateSDK()

Parameters:

None

Return Values:

Returns **PS_OK** if successful, for other values see PS errors list, p. 0

See Also:

PSInitializeSDK

Note:

Example:

See code example in p. 0

PSGetCameraManager

Description:

Returns **CameraManager** object. The object responses for interaction with cameras and for exchanging messages with them. The object provides list of connected cameras and their properties and opens camera control sessions.
To release memory from **CameraManager** instance, use **PSFree** command on **CameraManagerHandle**

Syntax:

CameraManagerHandle PSSDKAPI PSGetCameraManager()

Parameters:

None

Return Values:

CameraManager object or, if error occurs, nullptr.

See Also:

PSGetCameraList, **PSOpenSession**, **PSCloseSession**, **PSFree**

Note:

Example:

See code example in p. 0

PSCameraInitializationErrorSubscribe

Description:

Registers callback function for camera initialization error event.
Event is activated for errors which occur during the process of camera initialization, between the moment when camera is connected to the PC and a

moment when **PSCameraListChanged** event is activated. Errors which occur with initialized camera activate **PSCameraError** event.

Syntax:

PSSubscriberHandle PSSDKAPI PSCameraInitializationErrorSubscribe(PSCameraManagerHandle inCamMgr, PSCameraInitializationErrorCallback inCallback, void* inContext)

Parameters:

inCamMgr – current **CameraManagerHandle** object
inCallback – callback function to register. See **PSCameraInitializationErrorCallback** function description below.
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, PSCameraInitializationErrorCallback, PSCameraListChangedSubscribe, PSCameraErrorSubscribe

Note:

To unsubscribe from the event, call **PSFree** command for subscription handler.

Example:

PSCameraInitializationErrorCallback

Description:

Callback function for **PSCameraInitializationError** event. Function should be defined in client application and given to **PSCameraInitializationError Subscribe** function as an argument.

Syntax:

PSCameraInitializationErrorCallback(void* context, int code, wchar_t* camSystemId)

Parameters:

context – user data given to the SDK in **PSCameraInitializationErrorSubscribe** call
code – error code of **PSResult** type;
camSystemId – camera system identifier which initialization raised the error. This handler should be released by **PSFree** command

Return Values

None

See Also:

PSCameraListChangedSubscribe, **PSFree**

Note:

Example:

PSCameraListChangedSubscribe

Description:

Registers callback function for camera list changing event. Events contains information on their type (connection or disconnection a camera from the list)

Syntax:

SubscriberHandle PSCameraListChangedSubscribe(CameraManagerHandle, CameraListChangedCallback, void* inContext);

Parameters:

CameraManagerHandle - current **CameraManager** object
CameraListChangedCallback callback function to register. See its description below.
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, CameraListChangedCallback

Note:

Usual scenario of processing the events consists of re-reading camera list and checking whether the camera still connected to the PC or not. More complicated processing scenarios are also possible.

To unsubscribe from the event, call **PSFree** command for subscription handler

Example:

PSCameraListChangedCallback

Description:

Callback function for **PSCameraListChanged** event. Function should be defined in client application and given to **PSCameraListChangedSubscribe** function as an argument.

Syntax:

PSCameraListChangedCallback(void* context, PSCameraInfo* devInfo, int connectionState)

Parameters:

context – user data given to the SDK in **PSCameraListChangedSubscribe** call
devInfo – information about camera issued change in the camera list;
connectionState – camera connection state (see **PSCameraConnectionState**, p. 0)

Return Values:

None

See Also:

PSCameraListChangedSubscribe

Note:

Example:

PSFree

Description:

Removes object by given link from the memory. Could receive any object of PS-SDK

Syntax:

void PSSDKAPI
PSFree(void*)

Parameters:

PS-SDK object to release memory from

Return Values:

None

See Also:

Note:

The command would also works correctly if argument's value is nullptr.

Example:

See code example in p. 0

PSGetCameraList

Description:

Returns list of compatible cameras connected to the computer. Camera list is a collection of structure of **CameraInfo** type.

Syntax:

void PSSDKAPI
PSGetCameraList(PSCameraManagerHandle inCamMgr, PSCameraInfo outCameraList, int* outListSize)**

Parameters:

inCamMgr – current **CameraManager** object

Return Values:

outCameraList – list of **PSCameraInfo** items with descriptions of connected cameras. Memory should be released from the list after its using by **PSFree** command
outListSize – number of items in **outCameraList** list

See Also:

PSGetCameraManager

Note:

Camera list getting process is logged, so SDK log contains full list of cameras found in the system, including incompatible ones.
If there are no cameras connected to the PC at the moment, **outCameraList** is Null

Example:

See code example in p. 0

PSOpenSession

Description:

Opens session of camera remote control by **CameraId** from camera description

Syntax:

PSResult PSSDKAPI
PSOpenSession(**CameraManagerHandle inCamMgr, CameraId inCamId, SessionHandle* outSession**);

Parameters:

inCamMgr – current **CameraManager** instance
inCamId – unique id of the camera, of type **CameraId**

Return Values:

Returns **PS_OK** if successful, for other values see PS errors list, p. 37
outSession – Session handler object which is used for sending messages to the camera

See Also:

PSGetCameraManager, PSCloseSession, PSFree, PSGetCameraId

Note:

When session is opened, files list **FileList** is initialized. All files from camera's flash-card are removed and **FileList** after initialization is always empty**..**
To close session, call **PSFree** for session handler

Example:

See code example in p. 0

PSCameraErrorSubscribe

Description:

Registers callback function for camera error event.
Event is activated for errors which occur during work with the camera with opened session. Errors which occur during camera initialization process activate **PSCameraInitializationError** event.

Syntax:

PSSubscriberHandle PSSDKAPI
PSCameraErrorSubscribe (SessionHandle inSession,
PSCameraErrorCallback inCallback,
void* inContext)

Parameters:

inSession – current session opened for the camera
inCallback – callback function to register. See **PSCameraErrorCallback** function description below.
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, PSCameraErrorCallback, PSCameraInitializationErrorSubscribe

Note:

To unsubscribe from the event, call **PSFree** command for subscription handler.

Example:

PSCameraErrorCallback

Description:

Callback function for **PSCameraError** event. Function should be defined in client application and given to **PSCameraErrorSubscribe** function as an argument.

Syntax:

PSCameraErrorCallback(void* context, int code, wchar_t* camSystemId)

Parameters:

context – user data given to the SDK in **PSCameraInitializationErrorSubscribe** call
code – error code of **PSResult** type;
camSystemId – system identifier of the camera which raised the error. This handler should be released by **PSFree** command

Return Values

None

See Also:

PSCameraErrorSubscribe, **PSFree**

Note:

Example:

PSGetCameraInfo

Description:

Returns camera's info by its open session handler

Syntax:

PSResult PSSDKAPI
PSGetCameraInfo(PSSessionHandle inSession, PSCameraInfo* outInfo);

Parameters:

inSession – handler of the session in which the camera is connected;

Return Values:

PSResult error code
outInfo – camera information

See Also:

Note:

Example:

PSGetCameraConnectionState

Description:

Checks if the camera still connected in the session.

Syntax:

PSResult PSSDKAPI
PSGetCameraConnectionState(PSSessionHandle inSession, int* outCameraConnectionState);

Parameters:

inSession – handler of the session through which the camera is connected

Return Values:

PSResult error code
outCameraConnectionState – **PSCameraConnectionState** value (**PS_CS_CONNECTED** or **PS_CS_DISCONNECTED**)

See Also:

PSOpenSession

Note:

The command checks whether the session was closed after disconnection of the camera and after receiving corresponding event. If the camera is disconnected from the session without closing it, the session should be closed manually.
Check is necessary due to possibility of physical disconnection of a camera without closing its session.

Example:

PSSetPreviewState

Description:

Controls receiving of preview images from the camera

Syntax:

PSResult PSSDKAPI
PSSetPreviewState(PSSessionHandle inSession, int inState);

Parameters:

inSession – handler of the session through which the camera is connected;
inState – required preview state of **PSPreviewState** type (**PS_PREVIEW_ENABLED** or **PS_PREVIEW_DISABLED**)

Return Values:

PSResult error code

See Also:

PSGetPreviewState

Note:

Receiving preview images from a camera costs additional PC CPU consumption.

Example:

See code example in p. 0

PSGetPreviewState

Description:

Returns current status of receiving preview images from the camera

Syntax:

PSResult PSSDKAPI
PSGetPreviewState(PSSessionHandle inSession, int* outState);

Parameters:

inSession – handler of the session through which the camera is connected;

Return Values:

PSResult error code
outState – preview state of **PSPreviewState** type

(**PS_PREVIEW_ENABLED** or **PS_PREVIEW_DISABLED**)

See Also:

PSSetPreviewState.

Note: .

Example:

PSNewPreviewFrameSubscribe

Description:

Registers callback function for event of creating new preview image.

Syntax:

PSSubscriberHandle PSSDKAPI
PSNewPreviewFrameSubscribe(PSSessionHandle inSession,
PSGenericEventCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register. See **PSGenericEventCallback** description for details
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values:

Subscription handler or, If error occurred, nullptr

See Also:

PSGenericEventCallback

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

See code example in p. 0

PSGenericEventCallback

Description:

Callback function for generic events which doesn't receive additional arguments. Function should be defined in client application and given to event subscription function as an argument.

Syntax:

PSGenericEventCallback(void* context);

Parameters:

context – user data given to the SDK in event subscription command

Return Values:

None

See Also:

PSNewPreviewFrameSubscribe

Note:

Example:

See code example in p. 0

PSGetPreviewFrame

Description:

Receives preview frame from the camera

Syntax:

PSResult PSSDKAPI
PSGetPreviewFrame(PSSessionHandle inSession, void outBMPData, int* outDataSize);**

Parameters:

inSession – handler of the session through which the camera is connected

Return Values:

PSResult error code;
outBMPData – preview image data in BMP-file format (**BITMAPFILEHEADER**, **BITMAPINFOHEADER**, array of pixels). To release memory from array **PSFree** command should be used;
outDataSize – size of **outBMPData** in bytes

See Also:

Note:

Important! Before next image could be received by means of **PSGetPreviewFrame** command, memory should be released from previous preview frame by applying **PSFree** command on **outBMPData** array

Example:

See code example in p. 0

PSUpdateAEAF

Description:

Refresh autoexposition and autofocus states of the camera

Syntax:

PSResult PSSDKAPI
PSUpdateAEAF(PSSessionHandle inSession);

Parameters:

inSession – handler of the session through which the camera is connected;

Return Values:

PSResult error code

See Also:

Note:

Example:

See code example in p. 0

PSGetMaximumZoom

Description:

Returns maximum optical zoom value available for a camera.

Syntax:

PSResult PSSDKAPI
PSGetMaximumZoom(PSSessionHandle inSession, int* outMaxZoom);

Parameters:

inSession – handler of the session through which the camera is connected;

Return Values:

PSResult error code
outMaxZoom – maximum available zoom value

See Also:

PSGetZoom, PSSetZoom

Note:

Example:

See code example in p. 0

PSGetZoom

Description:

Returns current optical zoom value

Syntax:

PSResult PSSDKAPI
PSGetZoom(PSSessionHandle inSession, int* outZoom);

Parameters:

inSession – handler of the session through which the camera is connected;

Return Values:

PSResult error code
outZoom – current zoom value

See Also:

PSGetZoomMaximumValue, PSSetZoom

Note:

Example:

See code example in p. 0

PSZoomChangedSubscribe

Description:

Registers callback function for event of changing current zoom value

Syntax:

PSSubscriberHandle PSSDKAPI
PSZoomChangedSubscribe(PSSessionHandle inSession,
PSGenericEventCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register of **PSGenericEventCallback** type
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values:

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, **PSGenericEventCallback**

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

PSSetZoom

Description:

Sets zoom value

Syntax:

PSResult PSSDKAPI
PSSetZoom(PSSessionHandle inSession, int inZoom);

Parameters:

inSession – handler of the session through which the camera is connected
inZoom – new zoom value

Return Values:

PSResult error code

See Also:

PSGetZoomMaximumValue, PSGetZoom

Note:

The function sends to the camera command to change zoom value and finishes immediately. When zoom action is completed, **ZoomCompleted** event is activated. To subscribe on the event **PSZoomCompletedSubscribe** function is used.

Example:

See code example in p. 0

PSZoomCompletedSubscribe

Description:

Registers callback function for event of completing zoom changing action in the camera

Syntax:

PSSubscriberHandle PSSDKAPI
PSZoomCompletedSubscribe(PSSessionHandle inSession,
PSGenericEventCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register of **PSGenericEventCallback** type

inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values:

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, **PSGenericEventCallback**

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

See code example in p. 0

PSShoot

Description:

Takes a photo

Syntax:

PSResult PSSDKAPI
PSShoot(PSSessionHandle inSession)

Parameters:

inSession – handler of the session through which the camera is connected

Return Values:

PSResult error code

See Also:

Note:

Example:

See code example in p. 0

PSShootCompletedSubscribe

Description:

Registers callback function for event of finishing taking photo

Syntax:

PSSubscriberHandle PSSDKAPI
PSShootCompletedSubscribe(PSSessionHandle inSession,
PSShootCompletedCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also:

PSFree

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

See code example in p. 0

PSShootCompletedCallback

Description:

Callback function for **PSShootCompleted** event. Function should be defined in client application and given to **PSShootCompletedSubscribe** function as an argument.

Syntax:

PSShootCompletedCallback (void* context, PSFileInfo* resFileId)

Parameters:

context – user data given to the SDK in **PSShootCompletedSubscribe** call
resFileId – information about taken image file. This handler should be released afterwards with **PSFree** function

Return Values

None

See Also:

PSShootCompletedSubscribe, PSFree

Note:

Example:

See code example in p. 0

PSGetFileList

Description:

Returns list of image files taken during current session

Syntax:

PSResult PSSDKAPI
PSGetFileList(PSSessionHandle inSession, PSFileInfo outList, int* outListSize);**

Parameters:

inSession – handler of the session through which the camera is connected

Return Values

PSResult error code

outList – list of **PSFileInfo** items for all taken image files. The memory should be released from the list by **PSFree** command
outListSize – number of items in the list

See Also:

Note:

If there are no taken images in the camera, **outList** is Null

Example:

PSDownloadFile

Description:

Download image file from the memory card

Syntax:

PSResult PSSDKAPI
PSDownloadFile(PSSessionHandle inSession, PSFileId inFileId, int inRemoveFile);

Parameters:

inSession – handler of the session through which the camera is connected
inFileId – id of transferring file in filelist
inRemoveFile – if true, the file is automatically removed from camera memory after transferring the file to the PC, otherwise image file should be removed manually afterwards with **PSDeleteFile** command.

Return Values

PSResult error code

See Also:

Note:

The function sends download command to the camera and finishes immediately. When file download is completed then **DownloadCompleted** event is activated. **PSDownloadCompletedSubscribe** function creates subscription on the event. Full path to downloaded file is sent as **resPath** argument to the event callback function of **PSDownloadCompletedCallback** type.

Example:

PSDownloadFileTo

Description:

Downloads image file to given path

Syntax:

PSResult PSSDKAPI
PSDownloadFileTo(PSSessionHandle inSession, PSFileId inFileId, wchar_t* inFilePath, int inRemoveFile);

Parameters:

inSession – handler of the session through which the camera is connected
inFileId – id of transferring file in filelist
inFilePath – path for saving transferred file
inRemoveFile – if true, the file is automatically removed from camera memory after transferring the file to the PC, otherwise image file should be removed manually afterwards with **PSDeleteFile** command.

Return Values

PSResult error code

See Also:

PSGetFileList, PSDownloadFile, PSDeleteFile

Note:

The function sends download command to the camera and finishes immediately. When file download is completed then **DownloadCompleted** event is activated. **PSDownloadCompletedSubscribe** function creates subscription on the event. Full path to downloaded file is sent as **resPath** argument to the event callback function of **PSDownloadCompletedCallback** type.

Example:

See code example in p. 0

PSDownloadCompletedSubscribe

Description:

Registers callback function for event of file download completing

Syntax:

PSSubscriberHandle PSSDKAPI
PSDownloadCompletedSubscribe(PSSessionHandle inSession,
PSDownloadCompletedCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, PSGetFileList, PSDownloadFile, PSDownloadFileTo PSDeleteFile

Note:

To unsubscribe from event, **PSFree** command should be called for subscription handler.
Callback function should call **PSFree** command for **resPath** parameter

Example:

See code example in p. 0

PSDownloadCompletedCallback

Description:

Callback function for **PSDownloadCompleted** event. Function should be defined in client application and given to **PSDownloadCompletedSubscribe** function as an argument.

Syntax:

PSDownloadCompletedCallback (void* context, PSFileId fileId, wchar_t* resPath)

Parameters:

context – user data given to the SDK in **PSDownloadCompletedSubscribe** call
resFileId – downloaded file identifier
resPath – full path to the downloaded file. This handler should be released afterwards with **PSFree** function

Return Values

See Also:

PSDownloadCompletedSubscribe, PSFree

Note:

Example:

See code example in p. 0

PSDeleteFile

Description:

Deletes given file from the camera memory card

Syntax:

PSResult PSSDKAPI
PSDeleteFile(PSSessionHandle inSession, PSFileId inFileId)

Parameters:

inSession – handler of the session through which the camera is connected
inFileId – id of transferring file in filelist

Return Values

PSResult error code

See Also:

PSFree, PSGetFileList, PSDownloadFile, PSDownloadFileTo PSDeleteFile

Note:

Example:

PSGetFocus

Description:

Returns manual focus distance value

Syntax:

PSResult PSSDKAPI
PSGetFocus(PSSessionHandle inSession, int* outDistance);

Parameters:

inSession – handler of the session through which the camera is connected
outDistance – manual focus distance value

Return Values:

PSResult error code

See Also:

PSSetFocus

Note:

The function returns real manual focus distance value used by camera. Use this function to check focus value on camera and for checking real value after using **PSSetFocus** on **PSFocusChanged** event.

Example:

PSSetFocus

Description:

Sets manual focus distance value

Syntax:

PSResult PSSDKAPI
PSSetFocus(PSSessionHandle inSession, int inDistance);

Parameters:

inSession – handler of the session through which the camera is connected
inDistance – new manual focus distance value

Return Values:

PSResult error code

See Also:

PSGetFocus

Note:

The function used to set manual focus in desired distance value. Set distance value in millimeters, camera will move focus to closest supported value in current zoom position. Subscribe with **PSFocusChangedSubscribe** function to **PSFocusChanged** event and use **PSGetFocus** function for checking real distance value used in camera.

Example:

PSFocusShift

Description:

Move manual focus distance value by step

Syntax:

PSResult PSSDKAPI
PSFocusShift(PSSessionHandle inSession, int inShift);

Parameters:

inSession – handler of the session through which the camera is connected
inShift – shift of new focus distance value

Return Values:

PSResult error code

See Also:

PSGetFocus

Note:

The function use only supported manual focus distance values to move manual focus. Set desired shift in number of steps, set shift direction with positive or negative values: +1 value will move manual focus to next supported value, -1 will move to previous supported value.

Example:

PSFocusChangedSubscribe

Description:

Registers callback function for event of changing manual focus value

Syntax:

PSSubscriberHandle PSSDKAPI
PSFocusChangedSubscribe(PSSessionHandle inSession,
PSGenericEventCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register of **PSGenericEventCallback** type
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values:

Subscription handler or, if error occurs, nullptr

See Also:

PSFree, **PSGenericEventCallback**

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

PSGetPropertyList

Description:

Returns list of all properties available at the moment

Syntax:

PSResult PSSDKAPI
PSGetPropertyList(PSSessionHandle inSession, int outList, int* listLen);**

Parameters:

inSession – handler of the session through which the camera is connected

Return Values

PSResult error code
outList – list of available properties
listLen – length of **outList** list

See Also:

Note:

Notice that list contains only currently available properties. It could change in dependence of properties values and camera mode

Example:

See code example in p. 0

PSGetPropertyDesc

Description:

Returns description of given property

Syntax:

PSResult PSSDKAPI
PSGetPropertyDesc(PSSessionHandle inSession, int inProp, PSProp_Desc outDesc);**

Parameters:

inSession – handler of the session through which the camera is connected
inProp – property id

Return Values

PSResult error code
outDesc – property description of the **PSProp_Desc** type. This handler should be released by **PSFree** function

See Also:

Note:

Notice that resulting **PSProp_Desc** structure **availableValues** list contains only values available at the moment. The list could change in dependence of other properties values and from camera mode.

Example:

See code example in p. 0

PSGetPropertyData

Description:

Returns value of given property

Syntax:

PSResult PSSDKAPI
PSGetPropertyData(PSSessionHandle inSession, int inProp, int* outRes);

Parameters:

inSession – handler of the session through which the camera is connected
inProp – property id

Return Values

PSResult error code
outRes – current property value

See Also:

Note:

Example:

PSetPropertyData

Description:

Sets value for given property

Syntax:

PSResult PSSDKAPI
PSSetPropertyData(PSSessionHandle inSession, int inProp, int inVal);

Parameters:

inSession – handler of the session through which the camera is connected

inProp – property id.
inVal – new property value

Return Values

PSResult error code

See Also:

Note:

Example:

See code example in p. 0

PSGetPropertyName

Description:

Returns readable name for given property

Syntax:

PSResult PSSDKAPI
PSGetPropertyName(int inProp, char outName);**

Parameters:

inProp – property id

Return Values

PSResult error code
outName – property name. Handler's release is not necessary.

See Also:

Note:

Example:

PSGetPropertyValName

Description:

Returns readable name for given value of given property

Syntax:

PSResult PSSDKAPI
PSGetPropertyValName(int inProp, int inPropVal, char outName);**

Parameters:

inProp - property id
inPropVal – property value id

Return Values

PSResult error code
outName – property value name. Handler's release is not necessary.

See Also:

Note:

Example:

See code example in p. 0

PSPropertyListChangedSubscribe

Description:

Registers callback function for event of properties list change

Syntax:

PSSubscriberHandle PSSDKAPI PSPropertyListChangedSubscribe(PSSessionHandle inSession, PSGenericEventCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register of **PSGenericEventCallback** type
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also: **PSFree, PSGenericEventCallback**

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

PSPropertyChangedSubscribe

Description:

Registers callback function for event of property change

Syntax:

PSSubscriberHandle PSSDKAPI PSPropertyChangedSubscribe(PSSessionHandle inSession, PSPropertyChangedCallback inCallback, void* inContext);

Parameters:

inSession – handler of the session through which the camera is connected
inCallback – callback function to register of **PSPropertyChangedCallback** type
inContext – designate application information to be passed by means of the callback function. Any data needed for your application can be passed.

Return Values

Subscription handler or, if error occurs, nullptr

See Also: **PSFree, PSPropertyChangedCallback**

Note:

To unsubscribe from event, **PSFree** command should be called for resulting subscription handler.

Example:

PSPropertyChangedCallback

Description:

Callback function for **PSPropertyChanged** event. Function should be defined in client application and given to **PSPropertyChangedSubscribe** function as an argument.

Syntax:

PSPropertyChangedCallback(void* context, int prop);

Parameters:

context – user data given to the SDK in **PSPropertyChangedSubscribe** call
prop – identifier of changed property

Return Values

See Also:

PSPropertyChangedSubscribe, PSFree

Note:

Example:

Errors

As return values, PS-SDK APIs return error codes defined as follows.

For each API, the return values mainly used are identified based on API characteristics. However, the principal factors that actually caused the problems are specified as error codes. Thus, all error codes may be specified in return values.

Returned value	Description
PS_OK	Function finished successfully
PS_FAILED,	Function failed due to unspecified reason
PS_SESSION_ALREADY_OPENED,	Attempt to open new connection session with the camera has finished unsuccessfully because there is already a session established with the camera
PS_CAMERA_NOT_FOUND	Attempt to send a command to the camera (or establish connection with the camera) was unsuccessful because camera wasn't found
PS_RESULT_NOT_READY	Command hasn't returned processing result because it is not ready yet
PS_UNSUPPORTED_CAMERA_MODE	Attempt to switch the camera in certain operation mode has been failed, because such mode is not supported by the camera
PS_UNSUPPORTED_PROPERTY	Attempt to get or set certain camera property has been failed, because such property is not available for the camera
PS_UNSUPPORTED_PROPERTY_VALUE	Attempt to set certain value for the property has been failed because the value is not supported by the property.
PS_LOW_BATTERY_LEVEL	Battery level too low

Properties

Properties of camera and images objects can be retrieved and set by means of **PSGetPropertyData**, **PSSetPropertyData**, and other APIs.

For certain properties, if the target object is a camera, you can use the **PSGetPropertyDesc** API to get the properties that can currently be set. For details, see the description of **PSGetPropertyDesc**.

If the target object is an image, it has properties besides current settings values—specifically, properties that store settings values at the time the image was shot. Current property settings values are usually indicated, assuming you do not particularly need the previous values.

For the various properties there are, this section explains the objects they describe and what the properties mean.

Property Details

Properties are explained in the following format:

4.4.xx PropertyID – property id

Description: Explains the role of the property and how to work with it

Value: Indicates possible values for the property.
Values are expressed as decimals unless otherwise noted..

Note: Considerations when using the API.

SProp_ImageSize

Description:

Indicates image size selected in the camera. Actual resolution of each size depends on camera model.

Value:

Value	Description
PS_ImageSize_Large	Large
PS_ImageSize_Medium_1	Medium1
PS_ImageSize_Medium_2	Medium2
PS_ImageSize_Medium_3	Medium3
PS_ImageSize_Small	Small

Note:

For image size property additional information available on image width and height

PSProp_JpegQuality

Description:

Quality of jpeg photo-files taken by the camera

Value:

Value	Description
PS_JpegQuality_Superfine	Superfine
PS_JpegQuality_Fine	Fine
PS_JpegQuality_Normal	Normal

Note:

PSProp_ISOSpeed

Description:

Current ISO speed of the camera. Number of available ISO speeds dependent on

the camera.

Value:

Value	Description
PS_ISOSpeed_Auto (0)	Auto
PS_ISOSpeed_ISO_80 (80)	ISO 80
PS_ISOSpeed_ISO_100 (100)	ISO 100
PS_ISOSpeed_ISO_200 (200)	ISO 200
PS_ISOSpeed_ISO_400 (400)	ISO 400
PS_ISOSpeed_ISO_800 (800)	ISO 800
PS_ISOSpeed_ISO_1600 (1600)	ISO 1600

Note:

Values in brackets represent actual integer values.
When Auto ISO is defined, camera determines itself which ISO speed it should use in each case, in correspondence with current light conditions, shutter speed and so on.

PSProp_ShootingMode

Description:

Indicates camera shooting mode

Value:

Value	Description
PS_ShootingMode_Auto	Fully automatic mode when camera decides values of all shooting parameters
PS_ShootingMode_Program	Program mode. Camera automatically sets aperture value and shutter speed, but flash-mode and ISO-speed value are taken from corresponding properties
PS_ShootingMode_Manual	Fully manual mode.
PS_ShootingMode_AV	Aperture priority mode. Shutter speed is set automatically in dependence on aperture value
PS_ShootingMode_TV	Shutter priority mode. Aperture value is set automatically on dependence on defined shutter speed value

Note:

PSProp_WBmode

Description:

Indicates white balance mode

Value:

Value	Description
PS_WBmode_Auto	Auto
PS_WBmode_Day_Light	Day light
PS_WBmode_Cloudy	Cloudy
PS_WBmode_Tungsten	Tungsten
PS_WBmode_Fluorescent	Fluorescent

PS_WBmode_Fluorescent_H	Fluorescent H
PS_WBmode_Custom	Custom value

Note:

PSProp_Av

Description:

Indicates aperture value for manual and aperture priority shooting modes. Available values depends on camera model and current Zoom value

Value:

Value	Description
PS_Av_3_4 (34)	3.4
PS_Av_4_0 (40)	4.0
PS_Av_4_5 (45)	4.5
PS_Av_5_0 (50)	5.0
PS_Av_5_6 (56)	5.6
PS_Av_6_3 (63)	6.3
PS_Av_7_1 (71)	7.1
PS_Av_8_0 (80)	8.0

Note:

Values in brackets represent actual integer values.

PSProp_Tv

Description:

Indicates shutter speed for manual and shutter speed priority camera modes

Value:

Value	Description
PS_Tv_15sec (150000)	15"
PS_Tv_13sec (130000)	13"
PS_Tv_10sec (100000)	10"
PS_Tv_8sec (80000)	8"
PS_Tv_6sec (60000)	6"
PS_Tv_5sec (50000)	5"
PS_Tv_3sec2 (32000)	3"2
PS_Tv_2sec5 (25000)	2"5
PS_Tv_2sec (20000)	2"
PS_Tv_1sec6 (16000)	1"6
PS_Tv_1sec3 (13000)	1"3
PS_Tv_1sec (10000)	1"
PS_Tv_0sec8 (8000)	0"8
PS_Tv_0sec6 (6000)	0"6
PS_Tv_0sec5 (5000)	0"5
PS_Tv_0sec4 (4000)	0"4
PS_Tv_0sec3 (3000)	0"3
PS_Tv_4 (2500)	¼
PS_Tv_5 (2000)	1/5
PS_Tv_6 (1666)	1/6
PS_Tv_8 (1250)	1/8

PS_Tv_10 (1000)	1/10
PS_Tv_13 (769)	1/13
PS_Tv_15 (666)	1/15
PS_Tv_20 (500)	1/20
PS_Tv_25 (400)	1/25
PS_Tv_30 (333)	1/30
PS_Tv_40 (250)	1/40
PS_Tv_50 (200)	1/50
PS_Tv_60 (166)	1/60
PS_Tv_80 (125)	1/80
PS_Tv_100 (100)	1/100
PS_Tv_125 (80)	1/125
PS_Tv_160 (62)	1/160
PS_Tv_200 (50)	1/200
PS_Tv_250 (40))	1/250
PS_Tv_320 (31)	1/320
PS_Tv_400 (25)	1/400
PS_Tv_500 (20)	1/500
PS_Tv_640 (15	1/640
PS_Tv_800 (12	1/800
PS_Tv_1000 (10	1/1000
PS_Tv_1250 (8)	1/1250
PS_Tv_1600 (6)	1/1600
PS_Tv_2000 (5)	1/2000
PS_Tv_2500 (4)	1/2500

Note:

Values in brackets represent actual integer values.

PSProp_RedEyeMode

Description:

Indicates the state of Red-eye reducing mode of the flash

Target Object:

Value:

Value	Description
PS_RedEyeMode_On	On
PS_RedEyeMode_Off	Off

Note:

PSProp_ExposureComp

Description:

Indicates exposure compensation value

Value:

Value	Description
PS_ExposureComp_minus_2 (0)	-2
PS_ExposureComp_minus_1_66 (34)	-1.66
PS_ExposureComp_minus_1_33 (67)	-1.33
PS_ExposureComp_minus_1 (100)	-1

PS_ExposureComp_minus_0_66 (134)	-0.66
PS_ExposureComp_minus_0_33 (167)	-0.33
PS_ExposureComp_0 (200)	0
PS_ExposureComp_plus_0_33 (233)	+0.33
PS_ExposureComp_plus_0_66 (266)	+0.66
PS_ExposureComp_plus_1 (300)	+1
PS_ExposureComp_plus_1_33 (333)	+1.33
PS_ExposureComp_plus_1_66 (366)	+1.66
PS_ExposureComp_plus_2 (400)	+2

Note:

Values in brackets represent actual integer values.

PSProp_FlashComp

Description:

Indicates flash compensation

Value:

Value	Description
PS_FlashComp_minus_2 (0)	-2
PS_FlashComp_minus_1_66 (34)	-1.66
PS_FlashComp_minus_1_33 (67)	-1.33
PS_FlashComp_minus_1 (100)	-1
PS_FlashComp_minus_0_66 (134)	-0.66
PS_FlashComp_minus_0_33 (167)	-0.33
PS_FlashComp_0 (200)	0
PS_FlashComp_plus_0_33 (233)	+0.33
PS_FlashComp_plus_0_66 (266)	+0.66
PS_FlashComp_plus_1 (300)	+1
PS_FlashComp_plus_1_33 (333)	+1.33
PS_FlashComp_plus_1_66 (366)	+1.66
PS_FlashComp_plus_2 (400)	+2

Note:

Values in brackets represent actual integer values.

PSProp_MeteringMode

Description:

Indicates exposure metering mode

Value:

Value	Description
PS_MeteringMode_Evaluative	Evaluative
PS_MeteringMode_Spot	Spot
PS_MeteringMode_Center	Center

Note:

PSProp_FocusingZone

Description:

Indicates focusing mode

Value:

Value	Description
PS_FocusingZone_Auto	Auto
PS_FocusingZone_Normal	Normal
PS_FocusingZone_Macro	Macro
PS_FocusingZone_Super_Macro	Super Macro
PS_FocusingZone_Infinity	Infinity

Note:

PSProp_FlashMode

Description:

Indicates flash mode

Value:

Value	Description
PS_FlashMode_Auto	Auto
PS_FlashMode_On	On
PS_FlashMode_Off	Off

Note:

PSProp_BatteryLevel

Description:

Read-only property indicating current battery level

Value:

Value	Description
PS_BatteryLevel_1	1
PS_BatteryLevel_2	2
PS_BatteryLevel_3	3
PS_BatteryLevel_4	4
PS_BatteryLevel_DC	DC – camera is connected to direct current source

Note:

PS_ManualFocusMode

Description:

Property for switching Manual focus mode.

Value:

Value	Description
PS_ManualFocusMode_Off	Manual focus mode turned off, autofocusing mode used
PS_ManualFocusMode_On	Manual focus mode turned on, camera will not use autofocus, set distance value for focusing

Note:

Data Types Used by the APIs

Data types defined under PS-SDK are listed in **pssdk.h** and **psproperty.h** in C language format. This section introduces data types unique to PS-SDK that are used by PS-SDK APIs. For the most recent type definitions, see the header files.

PSProp_ValExtendedInfo

Additional information for property value.

For **PSProp_ImageSize** property values extended info contains structure **ImageSize** with width and height

PSProp_Desc

Property description:

- **int* availableValues** – list of values available for the property at the moment (in the exact order of values for the property as in property description);
- **PSProp_ValExtendedInfo* extendedValueInfo** – list of additional information for property values in the same order as in **availableValues**. If there is no additional information for value properties, **extendedValueInfo** is a nullptr;
- **Int availableValuesLength** – length of **availableValues** list (and of **extendedValueInfo** list if it is defined);
- **Int isReadOnly** – read-only marker. **PS_TRUE** for read-only properties and **PS_FALSE** for read-write ones.

PSCameraInfo

Camera information:

- **CameraId id** unique camera identifier;
- **wchar_t name[PS_MAX_NAME_LEN]** – camera model name (**PS_MAX_NAME_LEN** – 128 chars);
- **wchar_t systemId[PS_MAX_SYSTEMID_LEN]** – camera system id (**PS_MAX_SYSTEMID_LEN** – 256 chars)
- **wchar_t psmVersion[PS_MAX_PSMVER_LEN]** – version of camera PSM-firmware

PSFileInfo

File information:

- **FileId id** unique file id
- **char name[PS_MAX_FILE_NAME_LEN]** filename (_MAX_FILE_NAME_LEN – 128 chars)
- **int size** filesize in bytes;

PSCameraConnectionState

Camera connection state

- **PS_CS_CONNECTED** camera connected
- **PS_CS_DISCONNECTED** camera disconnected

PSPreviewState

Possible image preview states:

- **PS_PREVIEW_DISABLED** preview disabled,
- **PS_PREVIEW_ENABLED** preview enabled

PSBool

PS-SDK Boolean values representations

PS_FALSE - False,
PS_TRUE - True

5. Code samples

This sample code is written in C++. All samples represented in the document are fragments of the example file **SampleDlg.cpp** included in SDK distribution.

Initialization of the SDK and CameraManager

```
// CSampleDlg msession handlers
BOOL CSampleDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
 // Get file save path
 m_strSavePath = GetPath ();
 //--------------------------------
 // Dialog controls are initialized
 //--------------------------------
 UpdateControls ();
 //-------------
 // PS-SDK START
 //-------------
 PSResult psErr = PS_OK;
 psErr = PSInitializeSDK();
 if ( psErr != PS_OK )
 {
       goto camerr;
 }
 //----------------------------
 // Connection of camera device
 //----------------------------
 m_CamManager = PSGetCameraManager ();
 if (m_CamManager == 0)
 {
       MessageBox (_T("PSGetCameraManager failed"));
 }
 //----------------------------
 // A picture buffer is created
 //----------------------------
 m_pPreviewWnd = &m_PreviewLocation;
 if (!CreateDIBBuffer ())
 {
       MessageBox( "Error creating the picture buffer" );
       EndDialog( 0 );
       return      FALSE;
 }
 return TRUE;  // return TRUE  unless you set the focus to a control
camerr:
 CString strErr;
 if ( psErr != PS_OK )
```

```
 {
       strErr.Format(_T("ErrorCode = 0x%08X"), psErr);
 }
 else
 {
       strErr = _T("Unknown error");
 }
 MessageBox (strErr);
 EndDialog (1);
 return FALSE;
}
```

SDK termination

```
void CSampleDlg::CleanUp ()
{
 if (m_hReleaseEvent != 0)
 {
       PSFree (m_hReleaseEvent);
       m_hReleaseEvent = 0;
 }

 if (m_hDownloadEvent != 0)
 {
       PSFree (m_hDownloadEvent);
       m_hDownloadEvent = 0;
 }

 if (m_hPreviewFrameEvent != 0)
 {
       PSFree (m_hPreviewFrameEvent);
       m_hPreviewFrameEvent = 0;
 }

 if (m_hZoomEvent != 0)
 {
       PSFree (m_hZoomEvent);
       m_hZoomEvent = 0;
 }

 if (m_Session != 0)
 {
       PSFree (m_Session);
       m_Session = 0;
 }

 FreeDIBBuffer ();

 //--------------------------------------
 // End processing of PS-SDK is performed
 //--------------------------------------
 PSTerminateSDK();
```

Camera connection

```
void CSampleDlg::OnConnect()
{
 if (m_CamManager == 0)
 {
       m_bConnected = FALSE;
       UpdateData (FALSE);
```

```
      return;
}

if (m_hReleaseEvent != 0)
{
      PSFree (m_hReleaseEvent);
      m_hReleaseEvent = 0;
}

if (m_hDownloadEvent != 0)
{
      PSFree (m_hDownloadEvent);
      m_hDownloadEvent = 0;
}

if (m_hPreviewFrameEvent != 0)
{
      PSFree (m_hPreviewFrameEvent);
      m_hPreviewFrameEvent = 0;
}

if (m_hZoomEvent != 0)
{
      PSFree (m_hZoomEvent);
      m_hZoomEvent = 0;
}

if (m_Session != 0)
{
      PSFree (m_Session);
      m_Session = 0;
}

//----------------
// Get camera list
//----------------
int nCount = 0;
PSCameraInfo* pCamList = 0;
PSGetCameraList (m_CamManager, &pCamList, &nCount);

if (nCount == 0)
{
      AddInformationMessage (_T("No camera devices available"));

      m_bConnected = FALSE;
      UpdateData (FALSE);

      return;
}

if (pCamList == 0)
{
      MessageBox (_T("PSGetCameraList failed: empty outCameraList"));

      m_bConnected = FALSE;
      UpdateData (FALSE);

      return;
}

//--------------------
// Get selected camera
//--------------------
int index = 0;

PSCameraInfo camInfo;
```

```
 memcpy( &camInfo, (pCamList + sizeof(PSCameraInfo) * index),
       sizeof(PSCameraInfo) );

 CString strModelName;
 CString strSystemId;

#ifdef _UNICODE
 strModelName = camInfo.name;
 strSystemId = camInfo.systemId;
#else
 char pMBStringName[PS_MAX_NAME_LEN * 2];
 memset (pMBStringName, 0, sizeof (pMBStringName));
 ::WideCharToMultiByte(CP_ACP, 0, camInfo.name, -1, pMBStringName,
       sizeof (pMBStringName) - 1, 0, 0);
 strModelName = (const char *)pMBStringName;

 char pMBStringSystemId[PS_MAX_SYSTEMID_LEN * 2];
 memset (pMBStringSystemId, 0, sizeof (pMBStringSystemId));
 ::WideCharToMultiByte(CP_ACP, 0, camInfo.systemId, -1,
       pMBStringSystemId, sizeof (pMBStringSystemId) - 1, 0, 0);
 strSystemId = (const char *)pMBStringSystemId;
#endif
//     {
//           CString strMsg;
//           strMsg.Format (_T("Camera selected:\r\nid: %d\r\nname:
//                 %s\r\nsystemId: %s"),
//                 (int)camInfo.id, strModelName, strSystemId);
//           AddInformationMessage (strMsg);
//     }

 ASSERT(pCamList->id == camInfo.id);

 //---------------------------------------
 // Open session of camera remote control
 //---------------------------------------
 PSResult psErr = PSOpenSession (m_CamManager, camInfo.id, &m_Session);

 if (psErr != PS_OK)
 {
       CString strErr;
       strErr.Format(_T("PSOpenSession failed: ErrorCode = 0x%08X"),
             psErr);
       MessageBox (strErr);

       m_bConnected = FALSE;
       UpdateData (FALSE);

       return;
 }

 if (m_Session == 0)
 {
       CString strErr;
       strErr.Format(_T("PSOpenSession failed: Wrong session handle.
             ErrorCode = 0x%08X"), psErr);
       MessageBox (strErr);

       m_bConnected = FALSE;
       UpdateData (FALSE);

       return;
 }

 {
       CString strMsg;
       strMsg.Format (_T("New camera connected:\r\nid: %d\r\nname:
```

```
            %s\r\nsystemId: %s"),
            (int)camInfo.id, strModelName, strSystemId);
      AddInformationMessage (strMsg);
}

//-----------------------------------------------
// Set handler for PSShootCompletedEvent event
//-----------------------------------------------
s_ShootCompletedParam.session = m_Session;
s_ShootCompletedParam.strSavePath = m_strSavePath;
s_ShootCompletedParam.hwndDlg = GetSafeHwnd ();

m_hReleaseEvent = PSShootCompletedSubscribe (m_Session,
      (PSShootCompletedCallback) pfnShootCompletedCallback,
      (void*) &s_ShootCompletedParam);

if (!m_hReleaseEvent)
{
      MessageBox (_T("PSShootingCompletedSubscribe failed"));

      PSFree (m_Session);
      m_Session = 0;

      m_bConnected = FALSE;
      UpdateData (FALSE);

      return;
}

//-----------------------------------------------
// Set handler for PSDownloadCompletedEvent event
//-----------------------------------------------
s_DownloadCompletedParam.hwndDlg = GetSafeHwnd ();

m_hDownloadEvent = PSDownloadCompletedSubscribe (m_Session,
      (PSDownloadCompletedCallback) pfnDownloadCompletedCallback,
      (void*) &s_DownloadCompletedParam);

if (!m_hDownloadEvent)
{
      MessageBox (_T("PSDownloadCompletedSubscribe failed"));

      PSFree (m_Session);
      m_Session = 0;

      m_bConnected = FALSE;
      UpdateData (FALSE);

      return;
}

//------------------
// Set up properties
//------------------
UpdateProperty (PS_ImageSize,  m_cbImageSize);
UpdateProperty (PS_JpegQuality,      m_cbImageQuality);
UpdateProperty (PS_WBmode,           m_cbWhiteBalance);
UpdateProperty (PS_FlashMode,  m_cbFlashMode);
UpdateProperty (PS_RedEyeMode,       m_cbRedEye);
UpdateProperty (PS_FlashComp,  m_cbFlashComp);
UpdateProperty (PS_ISOSpeed,   m_cbISOSpeed);
UpdateProperty (PS_Av,               m_cbAV);
UpdateProperty (PS_Tv,               m_cbTV);
UpdateProperty (PS_ExposureComp, m_cbExposureComp);
UpdateProperty (PS_AFMode,           m_cbAFMode);
UpdateProperty (PS_FocusingZone, m_cbFocusingZone);
```

```
 UpdateProperty (PS_MeteringMode, m_cbMeteringMode);
 UpdateProperty (PS_ShootingMode, m_cbShootingMode);
 UpdateProperty (PS_BatteryLevel, m_cbBatteryLevel);

 //-------------------
 // Set up zoom slider
 //-------------------
 UpdateZoom ();

 //-----------------------------------------------
 // Set handler for PSZoomCompletedEvent event
 //-----------------------------------------------
 m_hZoomEvent = PSZoomCompletedSubscribe (m_Session,
       (PSGenericEventCallback) pfnZoomCompletedCallback,
       (void*) GetSafeHwnd ());

 if (!m_hZoomEvent)
 {
       MessageBox (_T("PSZoomCompletedSubscribe failed"));

       PSFree (m_Session);
       m_Session = 0;

       m_bConnected = FALSE;
       UpdateData (FALSE);

       return;
 }

 m_bConnected = TRUE;
 UpdateData (FALSE);

 m_btnPreview.SetFocus ();
}

void CSampleDlg::OnAEAF()
{
 if (m_Session == 0)
 {
       return;
 }

 //---------------------------------
 // Autoexposure and autofocus update
 //---------------------------------
 PSResult psErr = PSUpdateAEAF (m_Session);

 if (psErr != PS_OK)
 {
       CString strErr;
       strErr.Format(_T("PSUpdateAEAF failed: ErrorCode = 0x%08X"), psErr);
       MessageBox (strErr);
 }
}
```

Preview frames receiving switching on

```
void CSampleDlg::OnPreview()
{
 if (m_Session == 0)
 {
       return;
 }

 if (!m_hPreviewFrameEvent)
```

```
{
	//-------------------------------------------
	// Set handler for PSNewPreviewFrameEvent event
	//-------------------------------------------
	s_PreviewFrameParam.hwndDlg = GetSafeHwnd ();

	m_hPreviewFrameEvent = PSNewPreviewFrameSubscribe (m_Session,
		(PSGenericEventCallback) pfnNewPreviewFrameCallback,
	(void*) &s_PreviewFrameParam);

	if (!m_hPreviewFrameEvent)
	{
		MessageBox (_T("PSNewPreviewFrameSubscribe failed"));

		m_bPreviewEnabled = FALSE;
		UpdateData (FALSE);

		return;
	}

	PSResult psErr = PSSetPreviewState (m_Session, PS_PREVIEW_ENABLED);

	if (psErr != PS_OK)
	{
		CString strErr;
		strErr.Format(_T("PSSetPreviewState failed: ErrorCode =
			0x%08X"), psErr);
		MessageBox (strErr);
	}
}

m_bPreviewEnabled = TRUE;
UpdateData (FALSE);
}
```

Implementation of PSGenericEventCallback function

```
void CSampleDlg::pfnNewPreviewFrameCallback (void* context)
{
 if (!context)
 {
	ASSERT(FALSE);
	return;
 }

 const PREVIEW_FRAME_PARAM*	pParam	= (PREVIEW_FRAME_PARAM*)context;

 if (pParam->hwndDlg == NULL)
 {
	ASSERT(FALSE);
	return;
 }

 ::SendMessage (pParam->hwndDlg, CAMERAEVENT_GETFRAME_MESSAGE, 0, 0);
}
```

Receiving preview frame from a camera

```
LRESULT CSampleDlg::OnGetPreviewFrame(WPARAM, LPARAM)
{
 ASSERT_VALID (this);

 //------------------
 // Get preview frame
 //------------------
 void* pBMPData = NULL;
 int nDataSize = 0;
```

```
PSResult psErr = PSGetPreviewFrame(m_Session, &pBMPData, &nDataSize);

if (psErr != PS_OK)
{
      OnPreviewClose ();

      CString strErr;
      strErr.Format(_T("PSGetPreviewFrame failed: ErrorCode = 0x%08X"),
      psErr);
      MessageBox (strErr);

      return 1;
}

if (!pBMPData || nDataSize <= 0)
{
      ASSERT(FALSE);
      OnPreviewClose ();
      return 1;
}

//------------------------
// Load bitmap information
//------------------------
PBITMAPFILEHEADER pBmpFileheader = (PBITMAPFILEHEADER)pBMPData;
PBITMAPINFOHEADER pBmpInfoheader = (PBITMAPINFOHEADER)((LPBYTE)pBMPData +
      sizeof(BITMAPFILEHEADER));
LPBYTE                        bpPixel = (LPBYTE)pBMPData +
pBmpFileheader->bfOffBits;

ASSERT(pBmpInfoheader->biWidth == PREVIEW_WIDTH);
ASSERT(pBmpInfoheader->biHeight == PREVIEW_HEIGHT);
ASSERT(pBmpInfoheader->biPlanes == 1);
ASSERT(pBmpInfoheader->biBitCount == 32);
ASSERT(pBmpInfoheader->biCompression == BI_RGB);

ASSERT(m_PreviewSurface.vpBits != NULL);
memcpy(m_PreviewSurface.vpBits, bpPixel, pBmpInfoheader->biSizeImage);

//----------------
// Free SDK buffer
//----------------
PSFree (pBMPData);

//------------------------
// Display preview picture
//------------------------
ASSERT_VALID (m_pPreviewWnd);

CRect rectPreview;
m_pPreviewWnd->GetWindowRect(&rectPreview);
BOOL bStretch = TRUE;

HDC hdcDest = ::GetDC (m_pPreviewWnd->GetSafeHwnd ());
HDC hdcSrc = ::CreateCompatibleDC (hdcDest);
::SelectObject (hdcSrc, m_PreviewSurface.hBmp);
if (bStretch)
{
      ::StretchBlt(hdcDest,
            1, 1, rectPreview.Width () - 2, rectPreview.Height () - 2,
            hdcSrc,
            0, 0, PREVIEW_WIDTH, PREVIEW_HEIGHT,
            SRCCOPY );
}
else
{
```

```
        ::BitBlt(   hdcDest,
                            0, 0, PREVIEW_WIDTH, PREVIEW_HEIGHT,
                            hdcSrc,
                            0, 0,
                            SRCCOPY );
  }
  ::DeleteDC (hdcSrc);
  ::ReleaseDC (m_pPreviewWnd->GetSafeHwnd (), hdcDest);

  return 0;
}
```

Update current Zoom value

```
void CSampleDlg::UpdateZoom ()
{
  if (m_Session == 0)
  {
        return;
  }

  //--------------------
  // Set up zoom slider
  //--------------------
  int nZoomMax = 0;
  PSResult psErr = PSGetMaximumZoom (m_Session, &nZoomMax);

  if (psErr != PS_OK)
  {
        CString strErr;
        strErr.Format(_T("PSGetMaximumZoom failed: ErrorCode = 0x%08X"),
              psErr);
        MessageBox (strErr);

        UpdateData (FALSE);
        return;
  }

  int nZoomTickFrequency = (nZoomMax > 10) ? nZoomMax / 10 : 1;

  int nZoomValue = 0;
  psErr = PSGetZoom (m_Session, &nZoomValue);

  if (psErr != PS_OK)
  {
        CString strErr;
        strErr.Format(_T("PSGetZoom failed: ErrorCode = 0x%08X"), psErr);
        MessageBox (strErr);

        UpdateData (FALSE);
        return;
  }

  m_sliderZoom.SetRangeMax(nZoomMax);
  m_sliderZoom.SetPos(nZoomValue);
  m_sliderZoom.SetTicFreq(nZoomTickFrequency);

  UpdateData (FALSE);
}
```

Change Zoom value

```
void CSampleDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
```

```
 if (m_Session == 0)
 {
       CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
       return;
 }

 //------------
 // Change zoom
 //------------
 if (nSBCode == TB_ENDTRACK)
 {
       int nZoomValue = m_sliderZoom.GetPos();

       PSResult psErr = PSSetZoom (m_Session, nZoomValue);

       if (psErr != PS_OK)
       {
             CString strErr;
             strErr.Format(_T("PSSetZoom failed: ErrorCode = 0x%08X"),
                   psErr);
             MessageBox (strErr);
       }
 }
 CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

Shutter release

```
void CSampleDlg::OnRelease()
{
 if (m_Session == 0)
 {
       return;
 }

 //----------------
 // Release command
 //----------------
 PSResult psErr = PSShoot (m_Session);

 if (psErr != PS_OK)
 {
       CString strErr;
       strErr.Format(_T("PSRelease failed: ErrorCode = 0x%08X"), psErr);
       MessageBox (strErr);
 }
}
```

PSShootCompletedCallback implementation

```
void CSampleDlg::pfnShootCompletedCallback (void* context, PSFileInfo*
       resFileId)
{
 if (!context || !resFileId)
 {
       ASSERT(FALSE);
       return;
 }

 const SHOOT_COMPLETED_PARAM*  pParam      =
       (SHOOT_COMPLETED_PARAM*)context;

 if (pParam->session == 0)
 {
       ASSERT(FALSE);
```

```
		return;
	}

	if (pParam->hwndDlg == NULL)
	{
		ASSERT(FALSE);
		return;
	}

	CString strFileName;
#ifdef _UNICODE
	strFileName = resFileId->name;
#else
	char pMBStringFileName[PS_MAX_FILE_NAME_LEN * 2];
	memset (pMBStringFileName, 0, sizeof (pMBStringFileName));
	::WideCharToMultiByte(CP_ACP, 0, resFileId->name, -1, pMBStringFileName,
		sizeof (pMBStringFileName) - 1, 0, 0);
	strFileName = (const char *)pMBStringFileName;
#endif

	CString strMsg;
	strMsg.Format (_T("Release completed:\r\nid: %d\r\nname: %s\r\nsize: %d"),
		(int)resFileId->id, strFileName, (int)resFileId->size);
	s_CamInfo.strMessage = strMsg;
	::SendMessage (pParam->hwndDlg, CAMERAEVENT_INFO_MESSAGE, ::GetDlgCtrlID
(pParam->hwndDlg), LPARAM (&s_CamInfo));

	//-----------------------------------------
	// Get the name of the downloaded image file
	//-----------------------------------------
	CString strFilePath;
	strFilePath.Format (_T("%s%s"), pParam->strSavePath, strFileName);

//	CSaveDialog dlg = new COpenSaveDlg();
//	dlg.FileName = fi.Name;
//	dlg.Filter = "jpg files (*.jpg)|*.jpg|All files (*.*)|*.*";
//	dlg.RestoreDirectory = true;
//
//	if (dlg.ShowDialog() == DialogResult.OK)

	const int nWFilePathLen = strFilePath.GetLength ();
	wchar_t* pWFilePath = new wchar_t [nWFilePathLen + 1];
	memset (pWFilePath, 0, sizeof (wchar_t) * (nWFilePathLen + 1));

#ifdef _UNICODE
	lstrcpy (pWFilePath, (LPCTSTR)strFilePath);
#else
	::MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, (LPCTSTR)strFilePath, -1,
		pWFilePath, nWFilePathLen);
#endif

	strMsg.Format (_T("Try to download:\r\nId: %d\r\nFilePath: %s"),
		(int)resFileId->id, strFilePath);
	s_CamInfo.strMessage = strMsg;
	::SendMessage (pParam->hwndDlg, CAMERAEVENT_INFO_MESSAGE, ::GetDlgCtrlID
		(pParam->hwndDlg), LPARAM (&s_CamInfo));

	//--------------
	// Download file
	//--------------
	PSResult psErr = PSDownloadFileTo ((PSSessionHandle)pParam->session,
		(PSFileId)resFileId->id, (wchar_t*)pWFilePath, true);

	if (psErr != PS_OK)
	{
		CString strErr;
```

```
        strErr.Format(_T("PSDownloadFileTo failed: ErrorCode = 0x%08X"),
               psErr);
        AfxMessageBox (strErr);
 }

 delete [] pWFilePath;
}
```

PSDownloadCompletedCallback implementation

```
void CSampleDlg::pfnDownloadCompletedCallback (void* context, PSFileId
fileId, wchar_t* resPath)
{
 if (!context || !resPath)
 {
        ASSERT(FALSE);
        return;
 }

 const DOWNLOAD_COMPLETED_PARAM*     pParam      =
(DOWNLOAD_COMPLETED_PARAM*)context;

 if (pParam->hwndDlg == NULL)
 {
        ASSERT(FALSE);
        return;
 }

 CString strPath = (LPCWSTR) resPath;

 CString strMsg;
 strMsg.Format (_T("Download completed:\r\nId: %d\r\nPath: %s"),
(int)fileId, strPath);
 s_CamInfo.strMessage = strMsg;
 ::SendMessage (pParam->hwndDlg, CAMERAEVENT_INFO_MESSAGE, ::GetDlgCtrlID
(pParam->hwndDlg), LPARAM (&s_CamInfo));
}
```

Check for support of given property

```
BOOL CSampleDlg::IsPropertySupported (const int nPropID)
{
 if (m_Session == 0)
 {
        return FALSE;
 }

 //---------------------------------
 // Get list of available Properties
 //---------------------------------
 int* pPropList = NULL;
 int nPropListLen = 0;
 PSResult psErr = PSGetPropertyList (m_Session, &pPropList, &nPropListLen);

 if (psErr != PS_OK)
 {
        CString strErr;
        strErr.Format(_T("PSGetPropertyList failed: ErrorCode = 0x%08X"),
              psErr);
        MessageBox (strErr);
 }

 if (!pPropList)
 {
```

```
        return FALSE;
 }

 BOOL bPropEvailable = FALSE;

 for (int i = 0; i < nPropListLen; i++)
 {
        if (*((int*)pPropList + i) == nPropID)
        {
               bPropEvailable = TRUE;
               break;
        }
 }

 if (pPropList != NULL)
 {
        PSFree (pPropList);
 }

 return bPropEvailable;
}
```

Get property data

```
void CSampleDlg::UpdateProperty (const int nPropID, CComboBox& cb)
{
 if (m_Session == 0)
 {
        return;
 }

 if (cb.GetSafeHwnd() == NULL)
 {
        ASSERT(FALSE);
        return;
 }

 //------------------------------
 // Enable Property if supported
 //------------------------------
 if (!IsPropertySupported (nPropID))
 {
        cb.EnableWindow (FALSE);
        return;
 }

 //-------------------------------------
 // Get available values of the Property
 //-------------------------------------
 PSProp_Desc* pPropDesc = NULL;
 PSResult psErr = PSGetPropertyDesc (m_Session, nPropID, &pPropDesc);

 if (psErr != PS_OK)
 {
        CString strErr;
        strErr.Format(_T("PSGetPropertyDesc failed: ErrorCode = 0x%08X"),
psErr);
        MessageBox (strErr);
 }

 if(pPropDesc == NULL)
 {
        ASSERT(FALSE);
        return;
 }
```

```
 cb.Clear ();

 int i;
 for (i = 0; i < pPropDesc->availableValuesLength; i++)
 {
       int* pValue = ((int*)pPropDesc->availableValues) + i;

       char* pValueName = NULL;
       psErr = PSGetPropertyValName (nPropID, *pValue, &pValueName);

       if (psErr != PS_OK)
       {
             break;
       }

       cb.InsertString(i, CString ((LPCSTR) pValueName));
       cb.SetItemData(i, *pValue);
 }

 if (psErr != PS_OK)
 {
       CString strErr;
       strErr.Format(_T("PSGetPropertyValName failed: ErrorCode = 0x%08X"),
psErr);
       MessageBox (strErr);
 }

 //--------------------
 // Get Property value
 //--------------------
 int nValue = 0;
 psErr = PSGetPropertyData (m_Session, nPropID, &nValue);

 if (psErr != PS_OK)
 {
       CString strErr;
       strErr.Format(_T("PSGetPropertyData failed: ErrorCode = 0x%08X"),
psErr);
       MessageBox (strErr);
 }

 for (i = 0; i < (int) cb.GetCount(); i++)
 {
       if ((int) cb.GetItemData (i) == nValue)
       {
             cb.SetCurSel (i);
             break;
       }
 }

 cb.EnableWindow (!pPropDesc->isReadOnly);

 if (pPropDesc != NULL)
 {
       PSFree (pPropDesc);
 }
}
```