HERMES

Please report API bugs & documentation errors to support@norpix.com

NorPix

Table of Contents

Introduction

Hermes only works under Windows XP, Vista and 7. Previous versions (*i.e.* Windows 3.1/NT/95/98/ME/2000) are not supported. The Hermes library is now compiled with VC++ 2012 and uses WinSxS.

To use the Hermes API, include the "*hermes.h*" and link it with the "*hermesAPI.lib*" library. The *HermesAPI.dll* must be copied in the same folder as your executable. It could also be placed in the windows/system folder, but this is not recommended. The HermesAPI.lib is built under Visual Studio 2012 and thus may or may not work with a previous version of Visual Studio.

Please contact sales@norpix.com to get authorization codes, otherwise the Hermes API classes will not be instantiated.

Hermes .NET Wrapper

Although Hermes API is a C++ library, a .NET wrapper is also available to use Hermes API from other languages (Visual Basic.NE, C#, etc). Sections that are specific to the .NET wrapper are highlighted in blue, like this paragraph. The majority of this documentation refer to the native C++ implementation of the library. When available, the .NET function equivalent are noted. To use the wrapper, add the HermesNET.dll assembly to your project references and use the namespace "NorPix.HermesNET". Note that the other DLL must still be installed too, HermesNET.dll is only a wrapper around those. The wrapper uses the unicode version of the Hermes API (HermesAPIu.dll) so this is the dll that must be bundled with the wrapper dll.

Definitions

Here are some basic definitions of concepts used within the Hermes API and its documentation:

Grabber : Used to designate a camera, frame grabber or any such image capturing device.

Image width : The width, in pixels, of an image.

Image height : The height, in pixels, of an image.

Image bitdepth : The number of bits used to store a single pixel. For example, the value for monochrome images (256 shades) will be 8 bits. Standard BGR images will be 24 bits (8 bits x 3 channels).

Real image bitdepth : The number of bits holding relevant information on each channel. For example, a 16 bits monochrome camera often use 10 or 12 bits to store information. In that case, the Image BitDepth would be 16 and the real image bitdepth would be 10 or 12. In the case of BGR 48 bits image, each pixel is stored in 48 bits, allocating 16 bits per color.

Image format : This is the format in which the image information is stored. Ex : Monochrome, BGR, RGB, YUV422, etc. For a list and description of all supported format see "*HEnums.h*".

Files included in this distribution

bin\freeimage.dll
bin\libtiff.dll
bin\libjpeg.dll
bin\zlib.dll
bin\ijl20.dll
bin\EuresysJPEG.dll (1)
bin\EasyMs60.dll (1)
bin\ETools.dll (1)
Place these in the same folder as the application's executable. These libraries are used by the image exporters and sequence files and need to be bundled with your application.

(1) = You only need these if you want to support the *Euresys* frame grabbers)

DemoGrab.**
A demo project that uses the "grabber", "sequencer" et "viewer" modules. Every *dll* from the "bin" folder needs to be copied to the "Release" or "Debug" folders to run the demo.

DemoPlay.**
A demo project that uses the "sequencer" and "viewers" modules. Every *dll* from the "bin" folder needs to be copied to the "Release" or "Debug" folders to run the demo.

DemoSeqToImg.**
A demo project that uses "sequencer" and "image exporter" modules. Every *dll* from the "bin" folder needs to be copied to the "Release" or "Debug" folders to run the demo.

DemoSeqToAvi.**
A demo project that uses "sequencer" and "avi exporter" modules. Every *dll* from the "bin" folder needs to be copied to the "Release" or "Debug" folders to run the demo.

Program Files\Common Files\Norpix\devices.**
This folder contains the grabbers *dll*. They must be registered before use and need to be bundled with your application. Some grabbers have their own sub folder since they use manufacturer *dlls* which can not be downloaded or obtained normally.

doc\Hermes.pdf
The present Hermes API documentation.

doc\Drivers.pdf
Information on 3rd party drivers that need to be installed for the HermesAPI drivers to successfully register themselves.

Program Files\Common Files\Norpix\NpxGrabMan.dll
For the application to work, this file must be registered and needs to be bundled with your application.

Program Files\Common Files\Norpix\NpxGrabprop.dll
This file must be registered to access the *property pages* of any grabber and needs to be bundled with your application.

bin\FastAlgo.dll
This file must be installed in the same folder as *HermesAPI(u).dll* and needs to be bundled with your application.

bin\cfitsio.dll
This file must be installed in the same folder as *HermesAPI(u).dll* and needs to be bundled with your application.

bin\IppWrapper.dll
This file must be installed in the same folder as *HermesAPI(u).dll* and needs to be bundled with your application.

Program Files\Common Files\Norpix\ImgFeederFilter.ax
This file must be registered to enable AVI exporting and needs to be bundled with your application.

bin\NpxDebug.exe
This little program allows to trap debug messages sent to the system debugger, enabling display of the messages from drivers & Info Strings.

bin\SysInfo.exe
This program allows to easily retrieve the information needed to be granted authorization codes. (MAC address or HDD number) This also allows benchmarking of the hard disk drives speed.

bin\dll register.reg
When double-clicked on it, this optional file will add special instruction to the registry, allowing to simply double-click on *dll* files to register them. (This simply associate *regsvr32* as the default program used when double-clicking a *dll* file.)

bin\vcredist_x86 SP1.exe
Microsoft installer to install all the required WinSxS DLLs on client computers.

This file must be installed in the same folder as *HermesAPI(u).dll* and needs to be bundled with your application.

The following are the Hermes API files, as all headers are enclosed in it, only *Hermes.h* needs to be included to access all of the Hermes API. The correct library*, HermesAPI.lib* or *HermesAPIu.lib,* must also be linked to it.

Hermes.h	->Global Header
HermesAPI.lib	->Hermes Library
HermesAPIu.lib	->Hermes Unicode Library
HermesAPI.dll	->Hermes Runtime *dll*
HermesAPIu.dll	->Hermes Runtime Unicode *dll*
HGrabModule.h	->Grabber Module
HGrabCallbacks.h	->Grabber Module Callbacks
HImageExporter.h	->Image Exporter Module
HAviExporter.h	->AVI Exporter Module (obsolete, use HAviFile for new projects)
HSequence.h	->Sequence Module
HSequencerCallbacks.h	->Sequence Module Callbacks
HFPSMonitor.h	->FPS Monitor Module
HFPSMonitorCallbacks.h	->FPS Monitor Module Callbacks

HViewer.h	->Viewer Module
HEnums.h	->Miscellaneous Enumerations
HErrorManager.h	->Error Manager
HermesUtils.h	->Various utility functions
HImage.h	->Image Wrapper
HItem.h	->Items Used By The AVI Exporter Module
HBayerConverter.h	->Applies a bayer conversion on a mono image
HImageConverter.h	->Converts an image to a different format
HIODriver.h	->For IO operations
ImportExport.h	->Enables class imports
HAviFile.h	->AVI file playback & recording module
HAviCodecs.h	->Used by HAviFile
HAviCallbacks.h	->Avi module callbacks
MetadataID.h	->Unique identifiers for metadata
HMetadata.h	->The metadata class
NpxMetadata.h	->Native metadata classes
HMetadataInfo.h	->Metadata information
HMetadataManager.h	->Metadata Manager

Hermes.NET

The following files are related to .NET support. Hermes.NET is a wrapper around the Hermes API. If you can use both, it is strongly recommended to use the C++ API because the .NET wrapper will always be slower than the native implementation due to the type conversions overhead and memory management.

HermesNet\Release\HermesNET.dll
is the main DLL that must be added as a reference to your project. It is a wrapper around HermesAPIu.dll so the native DLLs mentioned earlier are still needed.

Copy this file to the same folder as the application .exe

Also bundled are various C# demos that show how to use the HermesNET. (mostly equivalent of the native demos)
HermesNET\DemoGrab.**
HermesNET\DemoPlay.**
HermesNET\DemoSeqToAVI.**
HermesNET\DemoSeqToImg.**

Windows Platform SDK & Direct X

It is recommended to install the latest Windows Platform SDK available for download at :
http://www.microsoft.com/msdownload/platformsdk/sdkupdate

Having the “DirectX 9.0 SDK Update” is also necessary. It can be downloaded at:
http://msdn.microsoft.com/downloads/

Unicode Support

If the application is *Unicode*, it must be linked to the Unicode version of the API.

	NON UNICODE	**UNICODE**
Library	*HermesAPI.lib*	*HermesAPIu.lib*
dll	*HermesAPI.dll*	*HermesAPIu.dll*

Unicode mode uses a specific method to store strings. Each character is stored on 16 bits instead of 8 bits. For instance, this allows Unicode applications to support languages like Japanese which have more than 256 characters in their "alphabet". To add Unicode support to the application, replace the Preprocessor definition “*_MBCS*” by “*UNICODE, _UNICODE*” in the project settings. The proper Unicode MFC libraries mus also be installed from the *Visual C++* CD-ROM.

IMPORTANT : If you use the unicode library with Visual C++ 2005 or greater : Open the project's Property Pages dialog box. In the C/C++ section, open the Language page and modify the "Treat wchar_t as Built-in Type" to "No". You will get linking errors if you don't set this properly.

For more information about Unicode, please check MSDN documentation (see TCHAR).

Hermes and multi-threading

All Hermes API classes functions can be called from different threads. However, it is recommended to use locks (mutex, semaphores or critical sections) to synchronize access from different threads. For example, you must not allow a thread to call HSequence::Close() while another thread is in HSequence::Write().

IMPORTANT : A large part of the API is built upon the COM architecture (every DLL that must be "registered" is a COM DLL). To allow the use of COM objects in its space, a thread must call CoInitialize(NULL) before using any COM objects and CoUninitialize() when has finished using them. Each CoInitialize must have a matching CoUninitialize.

Classes such as HGrabModule, HAVIExporter and HAviFile make extensive uses of COM objects (for grabbers & codecs). As those classes implicitly call CoInitialize(NULL) in their constructor, there is no need for the programmer to call it unless the class instance will be used from another thread. In such a case, CoInitialize should be the first function called upon thread creation.

Hermes in a Window NT Service

As the Hermes API was not designed to run as a Windows service, it should never be used as a part of such an application.

Registering drivers

Hermes grabber drivers are COM *dll*. To be usable, they have to be registered by the system.

An Hermes Driver *dll* can be registered in the command prompt by using the “regsvr32” command. For example, “r*egsvr32 NpxQimaging.dll*”. If registration is successful, the *HGrabModule Load* function could then be used to load the Hermes QImaging driver. The authorization code for *QImaging* along with compatible driver versions must also be at hand.

A dll registration failure could mean that it could not find one of the *dll* dependency. Missing files can be listed by using a dependency tool such as "*Depends*", which ships with Visual C++ 6.0 and greater. The *QImaging* camera, for instance, needs a Hermes called NpxQImaging.*dll*. To register correctly, it needs to have access to the QCAMDRIVER.*dll*, a *dll* that only installed trough the *QCaptureSuite* software (available from the manufacturer web site or on CD).

Each Hermes driver is meant to be used with a specific version of the grabber driver, usually the one provided by the grabber's manufacturer at the time of Hermes API release.

IMPORTANT : It is also necessary to register *NpxGrabman.dll* and *NpxGrabprop.dll* for API to connect to the drivers and display their property pages.

For the image exporter module to work correctly (HImageExporter), the following files need to be present in the same folder as the executable : *freeimage.dll, libtiff.dll, libjpeg.dll, zlib.dll.*

For the sequence module to work correctly (HSequence), the following files need to be present in the same folder as the executable : *ijl20.dll.*

If needed, register the ImageFeederFilter (*ImgFeederFilter.ax*) to enable the AVI Exporter module (*HAviExporter*).

Hermes API Protection

As most modules in the Hermes API are protected trying to use an API or driver for which the authorization are missing will return a "*H_ERROR_NOAUTHORIZATION*" error.

Please contact sales@norpix.com to obtain a registry file (*.npx*) in which the authorizations codes will be written. The npx files are actually renamed .reg files. This is done because a lot of e-mail system are blocking .reg attachments. But you can simply rename the file received to a .reg and double-clicking on it to write the authorization codes to the system registry so they can be found by the Hermes API runtimes.

4 authorization methods are available :

External USB Key

The major advantage of this kind of protection is that it allows to switch your codes from one system to another. In order for Hermes to read the license on the USB Key, a third party software driver must be installed on the computer. The USB Key needs to stay connected to the system's USB port while Hermes API runtimes are loaded.

Machine Code

This is the favored authorization method to be used if the external USB key method is not suitable. The machine code is built from a number of system parameters and can be retrieved by using SysInfo.exe.

MAC Address

The authorization code for a MAC address protection will allow the system with a matching MAC address to run the specified Hermes runtimes. The MAC address is an unique identifier specific to each Ethernet network card. To retrieve it, either use Hermes API's "sysInfo.exe", type "ipconfig /all" in the command prompt (the MAC address is called "Physical Address") or use the Windows Platform SDK "GetAdaptersInfo" function to retrieve it.

HDD s/n

The last protection method uses the serial number of the system primary hard disk drive (the drive where Windows is installed). However, should a primary hard disk failure, reformat or replacement occur, a new authorization code will be required.

Developer Environment Setup Check List

□ Download and install the latest Windows Platform SDK.

□ Download and install DirectX 9.0 SDK Update (release).

□ Follow the "User Environment Setup Check List" for the remaining steps.

User Environment Setup Check List

□ Make sure the user system is Windows XP or better.

□ Make sure the client has DirectX 9.0 or greater installed. Older versions might work but this has not been tested.

□ Install *HermesAPI.dll* (or HermesAPIu.*dll* for Unicode application) in the same folder as the application ".exe".

□ Install and register *NpxGrabman.dll* and *NpxGrabprop.dll.*

□ Install *FastAlgo.dll* & IppWrapper.dll in the same folder as the ".exe" application.

□ Install and register the drivers *dll* supported by the application. **Warning :** As the driver *dll* will refuse to register itself if it is not able to find all dependencies, manufacturer *dlls* must be installed prior to this step. A list of every supported grabber and its *dll*'s name is available in *HEnums.h.*

□ Install "*ijl20.dll*", "*freeimage.dll*", "*libtiff.dll*", "*libjpeg.dll*" and "*zlib.dll*" in the same folder as the ".exe" application.

□ Install the "vcredist_x86 SP1.exe".

□ If implementing the *HAviExporter*, install and register "*ImgFeederFilter.ax*" on the system.

□ If you use the .NET wrapper, install the latest .NET framework from WindowsUpdate.

□ Contact NorPix sales (sales@norpix.com) to get authorization codes for the user system. The favored protection method is by "MAC Address", then comes the USB Key and in last resort , the "HDD s/n" method. "MAC Address" and "HDD s/n" can be retrieved by using Hermes API's "SysInfo.exe".

Organization

The Hermes API is a small suite of modules :

Grabber Module : Assigned to a frame grabber, it capture frames in HImage wrappers.

Image Exporter Module : Export HImages to BMP, JPEG, TIFF, PNG, JPEG2000, FIT.

AVI Exporter Module : Exports HImages to AVI files. (obsolete, use HAviFile for new projects)

AVI Recording & Playback Module : Manage the recording and playback of AVI files.

Sequence Module : Reads and writes HImage to and from a sequence file (*.seq*) on disk by using *HsequencerDisk* or to RAM by using HSequencerRAM.

Viewer Module : Displays a HImage on a specified Device Context.

FPS Monitor Module : Computes the frame rate of any specified events.

The modules also use the following support classes :

HImage : Includes the data of a captured image.

HItem : Represents a single item of the lists returned by the HAviExporter module.

About time_t

time_t is used to represent an absolute time. In Hermes, the time stamps are always handled with 3 values :

-a "long" time : the time elapsed since midnight (00:00:00), January 1, 1970 (in seconds).

-a "unsigned short" timeMS : the millisecond part.

-a "unsigned short" timeUS : the microsecond part.

Hermes uses the 32-bits version of *time_t* time variable. It is considered as a "long" time to work around the linking problems arising with Visual Studio 2005 and higher, in which *time_t* is a 64 bits structure. For more information on the *time_t* structure, please refer to the MSDN Library.

General functions

SetErrorOutput

declaration :
void SetErrorOutput(bool debugString, bool visual, bool infoString)

void SetErrorOutput(bool debugString, bool visual, bool infoString)

description :
Most modules implement *SetErrorOutput,* allowing to set the module's behavior when encountering an error. By default, *debugString* and *infoString* are enabled and *visual* is disabled. DebugString and InfoString can be seen trough a program that hooks itself to the system debugger such as Hermes API's *NorpixDebug.exe*

parameters :
debugString [in] If true, when an error is detected, the error message will be automatically sent to the System Debugger.

visual [in] If true, when an error is detected, the error message will be automatically shown in a standard windows message box. It is recommended to set this to false for public releases.

infoString [in] If true, the information string will be sent to the System debugger. Information strings are status information related to the current process. Info strings are not errors.

return value :
None.

example :

```
HGrabModule module;
module.SetErrorOutput(true, true, true); //enable all reports.
```

GetLastError

declaration :
eHErrorCode GetLastError()

HEnums::ErrorCode GetLastError()

description :
Most modules implement *GetLastError*. Some functions will return *false* to indicate failure. When such a function fails, *GetLastError()* can be called to retrieve the error message related to the cause of the error.
For example, if *ModuleX.Foo()* returns *false*, immediately call *ModuleX.GetLastError()* to get the error code. This error code can be transmitted to the *HErrorManager::GetErrorString(eHErrorCode code)*. The returned value is a string describing the error. This is the same error message shown when the *debugString* and/or *visual* parameters of *SetErrorOutput()* are enabled.

parameters :
None.

return value :
The error code. Defined in "*HEnums.h*".

example :

```
SomeModule module;

if(!module.DoSomething())
{
	//function DoSomething returned false, indicating an error
	eHErrorCode error = module.GetLastError();

	//Use the error manager to get the description of the error
	printf(HErrorManager::GetErrorString(error)); //display error message
}
```

HGrabModule

The Hermes grabbing module purpose is to connect to the image capturing hardware and convert the images to a generic image format. (see *HImage*)

The typical (simplified) call order is :

```
HGrabModule module;          //Instantiates a grab module.
module.SetCallbacks(...);    //Prepare the callbacks.
module.Load(..., ...);       //Loads the grabber driver.
module.StartStreaming();     //Starts image capture.
...                          //Processes in callback functions.
module.StopStreaming();      //Stops image capture.
module.Unload();             //Unloads the grabber driver.
```

Load

declaration :
bool HGrabModule::Load(eHSupportedModule card, LPCTSTR appName, bool showError)

bool Grabber::Load(HEnums.SupportedGrabber card, string appName, bool showError)

description :
Load will initialize and setup the module for grabbing, loading the grabber's driver. This must be called before starting streaming. A list of all supported grabbers and their *dll* name can be found in *HEnums.h.*

parameters :
card [in] The identifier of the grabber driver that will be managed by the module. The list of indentifiers along with a description of each can be found in the “HEnums.h” file.

appName [in] A string including the name of your application. This name will be used to set a registry path where the grabber settings will be saved. It will have the form *“HKEY_CURRENT_USER\Software\appName\Grabman\”*.

showError [in] If true, Hermes will show an error message if the grabber fails to initialize properly.

return value :
True if the function is successful or *false* if the function failed. In case of failure, *GetLastError* can be called to retrieve the error code.

example :

```
HGrabModule GrabModule;

//Load the “Norpix virtual grabber” driver.
GrabModule.Load(H_NORPIX_VIRTUAL, “My Application”, true);
```

```
//Use the grabber
...

//Close the driver once you don't need it anymore
GrabModule.Unload();
```

Unload

declaration :
bool HGrabModule::Unload()

bool Grabber::Unload()

description :
Unload will uninitialize and close the grabber driver. The grabber driver can not be used after calling *Unload*, unless it is reloaded by calling *Load*.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call *GetLastError* to retrieve the error code.

example :

see HGrabModule::Load()

GetCurrentGrabber

declaration :
eHSupportedModule HGrabModule::GetCurrentGrabber()

HEnums::SupportedGrabber Grabber::GetCurrentGrabber()

description :
GetCurrentGrabber retrieves the identifier of the currently loaded grabber. For a complete list of all grabbers, see *HEnums.h.*

parameters :
card [out]

return value :
The identifier of the currently loaded grabber. If no grabber is loaded, the returned value will be H_UNKNOWN.

GetDriverCount

declaration :
unsigned int HGrabModule::GetDriverCount()

int Grabber::GetDriverCount()

description :
GetDriverCount retrieves the number of grabbers that are supported by the current version of the Hermes API. This function is often used in conjunction with *GetDriverInfo* to build a list or menu presenting the choices available to the user.

parameters :
None.

return value :
The number of supported grabbers.

GetDriverInfo

declaration :
LPCTSTR HGrabModule::GetDriverInfo(unsigned int index, eHSupportedModule* driverID)

string Grabber::GetDriverInfo(int index, ref HEnums.SupportedGrabber driverID)

description :
GetDriverInfo can be used to retrieve the name and grabber identifier of a specific grabber.

parameters :
index [in] The index of the grabber to get information on. Valid values range from 0 up to (GetDriverCount() - 1).

driverID [out] Return the identifier of the grabber required by the Load function.

return value :
The function will return a string holding the grabber name for the specified index. Asking for an out-of-bounds index will return NULL.

example :

```
HGrabModule GrabModule;
eHSupportedModule driverID;

//Enumerate all supported grabbers
for(int index=0; index < GrabModule.GetDriverCount(); index++)
{
	printf("Driver found : %s", GrabModule.GetDriverInfo(index, &driverID));
}
```

GetDriverInfo

declaration :
LPCTSTR HGrabModule::GetDriverInfo(eHSupportedModule driverID)

string Grabber::GetDriverInfo(HEnums.SupportedGrabber driverID)

description :
GetDriverInfo can be used to retrieve the name of the grabber by using its identifier.

parameters :
driverID [in] The identifier of the grabber.

return value :
The function will return a string holding the grabber name for the specified index. Asking for an invalid driverID will return NULL.

example :

```
HGrabModule GrabModule;

//Show the name of the virtual grabber
printf(GrabModule.GetDriverInfo(H_NORPIX_VIRTUAL));
```

StartStreaming

declaration :
bool HGrabModule::StartStreaming()

bool Grabber::StartStreaming()

description :
StartStreaming will start the streaming process, telling the grabber to start capturing images at the maximum possible speed. *Load* must first be called to initialize the frame grabber/camera before you can *StartStreaming*. Captured images can then be retrieved and processed by using the callback functions provided in the HGrabCallbacks class. While streaming is enabled, more resources will be required from the CPU and computer bandwidth.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StopStreaming

declaration :

bool HGrabModule::StopStreaming()

bool Grabber::StopStreaming()

description :
StopStreaming will stop the streaming process. The driver will tell the grabber to stop the capture loop and queuing images. Once streaming has stopped, callbacks from the grabbing modules will no longer be received. Stopping streaming while unnecessary will free up system resources for other tasks.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsStreaming

declaration :
bool HGrabModule::IsStreaming()

bool Grabber::IsStreaming()

description :
Retrieves the streaming state, *i.e.* if the camera is currently capturing images or not.

parameters :
None.

return value :
True if streaming is on, *false* if streaming is off.

SetCallbacks

declaration :
bool HGrabModule::SetCallbacks(HGrabCallbacks* userCallbacks, unsigned int userData=0, void* userPtr=NULL)

See the SetCallbacks...() functions.

description :
SetCallbacks allows to set the callback class used by the grabbing module. (see the *HGrabCallbacks* class)

parameters :
userCallbacks [in] The callback class to use. This will relate to one of the user *HGrabCallbacks*-derived classes.

userData [in] Use this parameter to pass a value to be used in your callback functions.

userPtr [in] Use this parameter to pass a pointer to be used in your callback functions.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :

```
class MyCallback : public HGrabCallbacks
{
public:
        virtual void OnImageReceived(HImage* image, unsigned int userData=0,
                                     void* userPtr=0)
        {
                printf("image received !");
        }
};

...

HGrabModule GrabModule;
MyCallback callback;

//Sets the callback class the grabber will use
GrabModule.SetCallback(&callback, 0, NULL);
```

ShowProperties

declaration :
bool HGrabModule::ShowProperties(HWND parentWndHandle)

bool Grabber::ShowProperties(IntPtr parentWindow)

description :
ShowProperties will display the property pages of the currently loaded frame grabber. The pages shown will change depending if the frame grabber is currently streaming or not.

parameters :
parentWndHandle [in] Handle to the parent window. If NULL, the dialog will not be modal.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetBufferCount

declaration :
void HGrabModule::SetBufferCount(unsigned int count)

void Grabber::SetBufferCount(uint count)

description :
SetBufferCount allows the user to specify the number of buffers to be used by the grabbing mechanism. Having a large number of buffers will help to prevent frame drops. However, having too many buffers will quickly eat the available RAM, so there is a compromise to be made. The default 5 buffers should be enough for most applications. *SetBufferCount* must be called before loading the frame grabber, as the buffer pool is created in the *Load* function.

parameters :
count [in] The number of buffers to create.

return value :
None.

GetBufferCount

declaration :
unsigned int HGrabModule::GetBufferCount()

uint Grabber::GetBufferCount()

description :
GetBufferCount retrieves the number of buffers used by the grabbing mechanism. The default value is 5, but can be changed with the *SetBufferCount* function.

parameters :
None.

return value :
The number of buffers used by the grabbing mechanism.

GetBufferUsage

declaration :
long HGrabModule::GetBufferUsage()

int Grabber::GetBufferUsage()

description :
This function returns the amount of frames waiting to be processed by the OnImageReceived callback. If this number reaches the total buffer count, the application will start dropping frames. The best place to call this function is in the OnImageReceived callback function.

parameters :
None.

return value :
The number of buffers waiting to be processed.

GetMaxBufferUsage

declaration :
long HGrabModule::GetMaxBufferUsage()

int Grabber::GetMaxBufferUsage()

description :
This function returns the maximum number of buffers reached since acquisition started (since StartStreaming()).

parameters :
None.

return value :
Maximum buffer usage since acquisition started.

GetTotalBufferCount

declaration :
long HGrabModule::GetTotalBufferCount()

int Grabber::GetTotalBufferCount()

description :
This function returns the total number of frames allocated for capture. If there was enough resources to allocate all buffers, this value should be equal to what was set in *SetBufferCount*.

parameters :
None.

return value :
Total number of frames allocated for acquisition.

GetIODriver

declaration :
HIODriver* GrabModule::GetIODriver()

IODriver Grabber::GetIODriver()

description :
GetIODriver retrieves a pointer to the grabber IO Driver. Check HIODriver for details on how to use that class.

parameters :
None.

return value :
The grabber IO driver.

GetImageWidth

declaration :
unsigned int HGrabModule::GetImageWidth()

uint Grabber::GetImageWidth()

description :
GetImageWidth retrieves the width of the images captured by the grabber.

parameters :
None.

return value :
This function will return image width of the captured images. If no grabber is selected or if the value can not be read from the hardware, the value will be 0.

GetImageHeight

declaration :
unsigned int HGrabModule::GetImageHeight()

uint Grabber::GetImageHeight()

description :
GetImageHeight retrieves the height of images captured by the grabber.

parameters :
None.

return value :
This function will return image height of the captured images. If no grabber is selected or if the value can not be read from the hardware, the value will be 0.

GetImageBitDepth

declaration :

unsigned int HGrabModule::GetImageBitDepth()

uint Grabber::GetImageBitDepth()

description :
GetImageBitDepth retrieves the bit depth of images captured by the grabber.

parameters :
None.

return value :
This function will return bit depth of the captured images. If no grabber is selected or if the value can not be read from the hardware, the value will be 0.

GetImageBitDepthReal

declaration :
unsigned int HGrabModule::GetImageBitDepthReal()

uint Grabber::GetImageBitDepthReal()

description :
GetImageBitDepthReal retrieves the real bit depth of images captured by the grabber.

parameters :
None.

return value :
The function will return the real bit depth of captured images. If no grabber is selected or if the value can not be read from the hardware, the returned value will be 0.

GetImageSizeBytes

declaration :
unsigned int HGrabModule::GetImageSizeBytes()

uint Grabber::GetImageSizeBytes()

description :
GetImageSizeBytes retrieves the size, in bytes, of each image captured by the grabber.

parameters :
None.

return value :
The function will return the size of captured images in bytes. If no grabber is selected or if the value can not be read from the hardware, the returned value is 0.

GetImageFormat

declaration :
eHImageFormat HGrabModule::GetImageFormat()

HEnums.ImageFormat Grabber::GetImageFormat()

description :
GetImageFormat retrieves the format of images captured by the grabber.

parameters :
None.

return value :
This function will return format of the captured images. If no grabber is selected or if the format can not be determined, the returned value will be H_IMAGE_UNKNOWN.

SetROI

declaration :
bool HGrabModule::SetROI(unsigned int offsetX, unsigned int offsetY, unsigned int sizeX, unsigned int sizeY)

bool Grabber::SetROI(uint offsetX, uint offsetY, uint sizeX, uint sizeY)

description :
SetROI allows to directly set the Region Of Interest. As the ROI can not be changed while the grabber is streaming, *StopStreaming* must be called before *SetROI*. Many grabbers will round the values to match their internal formats. Thus, it is strongly recommended to call *GetROI* immediately after setting it, to retrieve the exact ROI value set.

parameters :
offsetX [in] The horizontal offset of the region.
offsetY [in] The vertical offset of the region.
sizeX [in] The width of the region.
sizeY [in] The height of the region.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetROI

declaration :
bool HGrabModule::GetROI(unsigned int& offsetX, unsigned int& offsetY, unsigned int& sizeX, unsigned int& sizeY)

bool Grabber::GetROI(ref uint offsetX, ref uint offsetY, ref uint sizeX, ref uint sizeY)

description :
GetROI retrieves current position and dimension of the Region Of Interest.

parameters :
offsetX [out] The horizontal offset of the region.
offsetY [out] The vertical offset of the region.
sizeX [out] The width of the region.
sizeY [out] The height of the region.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetBinning

declaration :
bool HGrabModule::SetBinning(unsigned int value)

bool Grabber::SetBinning(uint value)

description :
SetBinning allows to directly set the binning factor of the camera, if supported. The binning can not be changed while the grabber is streaming so *StopStreaming* must be called *SetBinning*. Call *GetBinning* immediately after to retrieve the exact binning value set.

parameters :
value [in] The binning factor.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetBinning

declaration :
bool HGrabModule::GetBinning(unsigned int& value)

bool Grabber::GetBinning(ref uint value)

description :
GetBinning retrieves current binning factor of the camera.

parameters :
value [out] The binning factor.

return value :

True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetMaxResolution

declaration :
bool HGrabModule::GetMaxResolution(unsigned int& maxSizeX, unsigned int& maxSizeY)

bool Grabber::GetROI(ref uint maxSizeX, ref uint maxSizeY)

description :
GetMaxResolution retrieves the maximum ROI size that can be currently set.

parameters :
maxSizeX [out] The maximum allowed width value.
maxSizeY [out] The maximum allowed height value.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsCOMPortAvailable

declaration :
bool HGrabModule::IsCOMPortAvailable()

bool Grabber::IsCOMPortAvailable()

description :
This function allows to check availability of the *Serial Communication interface*. This interface is used with frame grabbers to provide access to asynchronous serial reading and writing functions in the camera's API, via COM port or a CLSER***.*dll*, where *** is specific to the frame grabber vendor.

parameters :
None.

return value :
True if Serial Communication interface is available or *false* otherwise.

GetCommunicationType

declaration :
bool GetCommunicationType(long* comType)

bool Grabber::GetCommunicationType(ref int comType)

description :
GetCommunicationType retrieves the communication type.

It can be one of the following values:
0 - Communication is disabled
1 - via camera's API
*2 - via CLSER***.dll*
3 - via COM Port

parameters :
comType [out] The communication type.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Transmit

declaration :
bool Transmit(void* buffer, unsigned long* bufferSize, unsigned long serialTimeout)

bool Grabber::Transmit(System.IntPtr buffer, ref uint bufferSize, uint serialTimeout)

description :
Transmit allows to send a buffer to the camera.

parameters :
buffer [in] The buffer to be transmitted.
bufferSize [in/out] Contains the buffer size, indicating the maximum number of bytes to be written. Upon a successful call, *bufferSize* contains the number of bytes written.
serialTimeout [in] Indicates the timeout in milliseconds.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Receive

declaration :
bool Receive(void* buffer, unsigned long* bufferSize, unsigned long serialTimeout)

bool Grabber::Receive(System.IntPtr buffer, ref uint bufferSize, uint serialTimeout)

description :
This function retrieves the camera response.

parameters :

buffer [out] Points to a user-allocated buffer. Upon a successful call, *buffer* contains the data read from the serial device. Caller should ensure that *buffer* is at least *bufferSize* in size.
bufferSize [in/out] Contains the buffer size indicating the maximum number of bytes that the buffer can accommodate. Upon a successful call, *bufferSize* contains the number of bytes read successfully from the camera.
serialTimeout [in] Indicates the timeout in milliseconds.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetEndChar

declaration :
bool GetEndChar(long* endChar)

bool Grabber::GetEndChar(System.IntPtr endChar)

description :
GetEndChar retrieves the command end character index.

parameters :
endChar [out] The end character index.

It can be one of the following values:
0 - NULL character
1 - <CR>
2 - <LF>
3 - <CR LF>

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetTextMode

declaration :
bool GetTextMode(long* textMode)

bool Grabber::GetTextMode(System.IntPtr textMode)

description :
GetTextMode retrieves the current command format.

parameters :
textMode [out] The command format.

It can be one of these two values:

0 - ASCII
1 - Hexadecimal

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetProgFolder

declaration :
bool GetProgFolder(LPTSTR pathName)

bool Grabber::GetProgFolder(ref string pathName)

description :
GetProgFolder retrieves the folder path for program files. As the user can write scripting files containing commands, he can select the specific folder where these files will be located.

parameters :
pathName [out] The user defined folder path.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetSavedProgFolder

declaration :
bool GetSavedProgFolder(LPTSTR pathName)

bool Grabber::GetSavedProgFolder(ref string pathName)

description :
GetSavedProgFolder retrieves the last program folder path.

parameters :
pathName [out] The last program folder path.

return value :
True if the function is successful or false if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsGPSSupported

declaration :
bool IsGPSSupported()

description :
IsGPSSupported checks if the selected time source card has GPS capabilities.

parameters :
None.

return value :
True if the currently selected time source card has GPS capabilities or *false* otherwise.

GetGPSData

declaration :
bool GetGPSData(CGPSDataStruct* gpsData, bool readNow = false)

description :
GetGPSData retrieves the GPS data (latitude, longitude, altitude) from the time source card.

parameters :
gpsData [in/out] A valid pointer to a CGPSDataStruct object has to be passed. After a successful call, this parameter will hold the GPS data that was retrieved.

readNow [in] When set to *true,* the GPS data is retrieved right away. If set to *false(*default*)*, the first call to this function will start a thread that updates the GPS data every 250ms.

return value :
True if the function is successful or false if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StopReadingGPSData

declaration :
bool StopReadingGPSData()

description :
StopReadingGPSData stops the GPS data monitoring that was initiated when the GetGPSData function was called the first time.

parameters :
None.

return value :
True if the function is successful or false if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

.NET Callback Mechanism (.NET only)

The following delegates are used to create the callback functions.

public delegate void OnImageNotificationDelegate(HermesImage image, int userData, IntPtr userPtr);

public delegate void OnNotificationDelegate(int userData, IntPtr userPtr);

SetCallbackData

declaration :
void Grabber::SetCallbackData(unsigned int data)

description :
This function set the user data member than will be returned as a parameter of the callback functions.

parameters :
userData [in] An unsigned integer.

return value :
None.

SetCallbackPtr

declaration :
void Grabber::SetCallbackPtr(IntPtr ptr)

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
userPtr [in] A pointer.

return value :
None.

SetCallbackOnBeforeQueuingGrab

declaration :
void Grabber::SetCallbackOnBeforeQueuingGrab(OnNotificationDelegate func)

description :
This function set the callback function called before a free buffer gets queued for capture.

parameters :

func [in] The callback function.

return value :
None.

SetCallbackOnAfterQueuingGrab

declaration :
void Grabber::SetCallbackOnAfterQueuingGrab(OnNotificationDelegate func)

description :
This function set the callback function called after a free buffer has been queued for capture.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnAfterStartStreaming

declaration :
void Grabber::SetCallbackOnAfterStartStreaming(OnNotificationDelegate func)

description :
This function set the callback function called when the camera is live (streaming).

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnAfterStopStreaming

declaration :
void Grabber::SetCallbackOnAfterStopStreaming(OnNotificationDelegate func)

description :
This function set the callback function called when the camera is not live (not streaming).

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnBeforeStartStreaming

declaration :
void Grabber::SetCallbackOnBeforeStartStreaming(OnNotificationDelegate func)

description :
This function set the callback function called when ...

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnBeforeStopStreaming

declaration :
void Grabber::SetCallbackOnBeforeStopStreaming(OnNotificationDelegate func)

description :
This function set the callback function called when ...

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnBeforeImageReceived

declaration :
void Grabber::SetCallbackOnBeforeImageReceived(OnNotificationDelegate func)

description :
This function set the callback function called when waiting for an image to be captured.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnImageReceived

declaration :
void Grabber::SetCallbackOnImageReceived(OnImageNotificationDelegate func)

description :
This function set the callback function called everytime an image is captured.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnImageReleased

declaration :
void Grabber::SetCallbackOnImageReleased(OnNotificationDelegate func)

description :
This function set the callback function called when the buffer holding a captured image is made available to the capture mechanism again.

parameters :
func [in] The callback function.

return value :
None.

Settings & Adjustments

Remarks

The function *HGrabModule::ShowProperties* is the official function to access and change the settings of the camera / frame grabber. However, a few customers have expressed their desire to control these settings without having to use the *ShowProperties* GUI. The following functions were added to provide direct access to the driver adjustments and settings.

The user must be very careful when using these functions, as a lot of the error checking normally done in ShowProperties will not be implemented here. The number of settings available will also vary depending on the camera's streaming state.

As many drivers do not support these functions yet, it is possible to verify grabber support by opening the ShowProperties dialog and looking for tabs called "Adjustments" and "Settings". Everything displayed on the subsequent pages can be controlled with the following functions. The absence of the "Adjustments" or "Settings" pages means that the driver has not yet been updated to support these functions. If this is the case, please contact NorPix to ask for this particular driver to be prioritized in the porting list.

Also note that some of the settings can not be changed while the camera is streaming. For instance, a setting that would the current image format (mono/color/etc) can not be called while the camera is streaming as it would probably crash the driver along with Hermes. As settings differ from one driver to the other, an accurate list of all available settings and capabilities is unlikely . What can be done is better left for the user to experiment. As a general rule, everything in the "Settings" page can not be changed while the camera is streaming and everything in the "Adjustments" page can be changed regardless of the streaming state.

Since only few drivers are currently supporting these functions, *ShowProperties* should be prioritized as is the safest way to access the adjustments and settings.

Refer to "DemoGrab" for a more complete code sample.

GetSettingsCount

declaration :
bool HGrabModule::GetSettingsCount(long* count)

bool Grabber::GetSettingsCount(ref int count)

description :
This function retrieves the number of settings currently available. For a few drivers, this number of settings can change depending on the camera's streaming status.

parameters :
count [out] The number of settings available.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetSettingsCaps

declaration :
bool HGrabModule::GetSettingsCaps(long index, LPTSTR name, long* featureID)

bool Grabber::GetSettingsCaps(int index, ref string name, ref int featureID)

description :
This function retrieves the name and the "feature identifier" of each setting. This identifier is used by all other functions to refer to a specific setting.

parameters :
index [in] The index of the settings to retrieve. Value must be between 0 and (*SettingsCount* - 1).

name [out] The name of the feature. (*ex:* "Image format" or"Bit Depth") Allocate at least "TCHAR name[100];" to allocate enough space to write it in.

featureID [out] The feature identifier.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetValuesCount

declaration :
bool HGrabModule::GetValuesCount(long featureID, long* count)

bool Grabber::GetValuesCount(int featureID, ref int count)

description :
This function retrieves the number of available choices for a setting.

parameters :
featureID [in] The feature identifier of the setting.

count [out] The number of available choices.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetValuesCaps

declaration :
bool HGrabModule::GetValuesCaps(long featureID, long index, LPTSTR name, long* valueID)

bool Grabber::GetValuesCaps(int featureID, int index, ref string name, ref int valueID)

description :
This function retrieves the name and value identifier of a selectable choice.

parameters :
featureID [in] The identifier of the parent settings.

index [in] The index of the value to retrieve. Value must be between 0 and (*ValueCount* - 1).

name [out] The name of the value (*ex:* Mono, BGR, YUV). Allocate at least "TCHAR name[100];" to allocate enough space to write it in.

valueID [out] The value identifier.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetCurrentValues

declaration :
bool HGrabModule::GetCurrentValues(long featureID, long* value)

bool Grabber::GetCurrentValues(int featureID, ref int value)

description :
This function retrieves the currently selected choice for a particular setting.

parameters :
featureID [in] The feature identifier for which to retrieve the current choice.

value [out] The value identifier.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetCurrentValues

declaration :
bool HGrabModule::SetCurrentValues(long featureID, long value)

bool Grabber::SetCurrentValues(int featureID, int value)

description :
This function selects a new choice for a given setting.

parameters :
featureID [in] The feature identifier for which to set a new choice.

value [in] The value identifier.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAdjustmentCount

declaration :
bool HGrabModule::GetAdjustmentCount(long* count)

bool Grabber::GetAdjustmentCount(ref int count)

description :
This function retrieves the number of currently available adjustments. For a few drivers, this number of settings can change depending on the camera's streaming status.

parameters :
count [out] The number of adjustments available.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAdjustmentCaps

declaration :
bool HGrabModule::GetAdjustmentCaps(long index, long* featureID, LPTSTR name, long* min, long* max, double* dMin, double* dMax, BOOL* integer, BOOL* logSlider, BOOL* manual, BOOL* automatic, BOOL* oneShot)

bool Grabber::GetAdjustmentCaps(int index, ref int featureID, ref string name, ref int min, ref int max, ref double dMin, ref double dMax, ref bool integer, ref bool logSlider, ref bool manual, ref bool automatic, ref bool oneShot)

description :
This function retrieves the name and the "feature identifier" of each adjustment. The identifier is used in all other functions to reference the adjustment interacted with.

parameters :
index [in] The index of the adjustment to retrieve Value must be between 0 and (AdjustmentsCount - 1).

featureID [out] The feature identifier.

name [out] The name of the feature. (ex: "Gain" or "Exposure") Allocate at least "TCHAR name[100];" to allocate enough space to write it in.

min [out] If the adjustment accepts integer values, this is the minimum value accepted.

max [out] If the adjustment accepts integer values, this is the maximum value accepted.

dMin [out] If the adjustment accepts double values, this is the minimum value accepted.

dMax [out] If the adjustment accepts double values, this is the maximum value accepted.

integer [out] If TRUE, the adjustment accepts integer values. If FALSE it accepts double values.

logSlider [out] TRUE if the use of a logarithmic slider is recommended to set the value. (This is often the case with exposure time)

manual [out] TRUE if the value can be changed by the user using *SetIntegerAdjustment* or *SetDoubleAdjustment*. If FALSE, the user can not specify a value.

automatic [out] TRUE if the adjustment has an automatic mode.

oneShot [out] TRUE if the adjustment has a One-Shot capability.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetIntegerAdjustment

declaration :
bool HGrabModule::GetIntegerAdjustment(long featureID, long* value)

bool Grabber::GetIntegerAdjustment(int featureID, ref int value)

description :
This function retrieves the current value of a particular adjustment. The adjustment must accept integer values for this function to work.

parameters :
featureID [in] The feature identifier of the adjustment.

value [out] The current value.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetIntegerAdjustment

declaration :
bool HGrabModule::SetIntegerAdjustment(long featureID, long value)

bool Grabber::SetIntegerAdjustment(int featureID, int value)

description :
This function sets the value of a particular adjustment. The adjustment must accept integer values for this function to work. Only use if "manual" from GetAdjustmentCaps is TRUE.

parameters :
featureID [in] The feature identifier of the adjustment.

value [in] The new value. Must be between [min, max] read from *GetAdjustmentCaps*.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetDoubleAdjustment

declaration :
bool HGrabModule::GetDoubleAdjustment(long featureID, double* value)

bool Grabber::GetDoubleAdjustment(int featureID, ref double value)

description :
This function retrieves the current value of a particular adjustment. The adjustment must accept double values for this function to work.

parameters :
featureID [in] The feature identifier of the adjustment.

value [out] The current value.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetDoubleAdjustment

declaration :
bool HGrabModule::SetDoubleAdjustment(long featureID, double value)

bool Grabber::SetDoubleAdjustment(int featureID, double value)

description :
This function sets the value of a particular adjustment. The adjustment must accept double values

for this function to work. Only use if "manual" from GetAdjustmentCaps is TRUE.

parameters :
featureID [in] The feature identifier of the adjustment.

value [in] The new value. Must be between [dMin, dMax] read from *GetAdjustmentCaps*.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAutomaticAdjustment

declaration :
bool HGrabModule::GetAutomaticAdjustment(long featureID, BOOL* automatic)

bool Grabber::GetAutomaticAdjustment(int featureID, ref bool automatic)

description :
This function retrieves the mode of the adjustment. When in automatic mode, the value of the adjustment can change during capture without notifying the user. While in automatic mode, you can not change the current value by a call to SetXXXAdjustment. Only use if "automatic" from GetAdjustmentCaps is TRUE.

parameters :
featureID [in] The feature identifier of the adjustment.

automatic [out] If TRUE, the adjustment is currently in automatic mode.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetAutomaticAdjustment

declaration :
bool HGrabModule::SetAutomaticAdjustment(long featureID, BOOL automatic)

bool Grabber::SetAutomaticAdjustment(int featureID, bool automatic)

description :
This function sets the mode of the adjustment. Only use if "automatic" from GetAdjustmentCaps is TRUE.

parameters :
featureID [in] The feature identifier of the adjustment.

automatic [in] TRUE to enable automatic mode, FALSE to deactivate the automatic mode.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

OneShotAdjustment

declaration :
bool HGrabModule::OneShotAdjustment(long featureID)

bool Grabber::OneShotAdjustment(int featureID)

description :
This function triggers a "One-shot". One instance of a "One-Shot" function is a function that adjusts the camera exposure using the next image received, making no further exposure change to the next images. Only use if "oneShot" from GetAdjustmentCaps is TRUE.

parameters :
featureID [in] The feature ID.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsSupportAdvanceSettings

declaration :
bool HGrabModule::IsSupportAdvanceSettings()

bool Grabber::IsSupportAdvanceSettings()

description :
Check if the grab support advanceSettings.

parameters :
None.

return value:
True if the function is support advanceSettings.

DoModalAdvanceSettingsWindow

declaration :
bool HGrabModule::DoModalAdvanceSettingsWindow()

bool Grabber::DoModalAdvanceSettingsWindow()

description :
Open Modal Advance Settings Window.

parameters :
None.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

CreateAdvanceSettingsWindow

declaration :
bool HGrabModule::CreateAdvanceSettingsWindow(HWND parent)

bool Grabber::CreateAdvanceSettingsWindow(IntPtr parentWindow)

description :
Create a child style Advance Settings Window.

parameters :
parent [in] Parent window handle.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ReSizeAdvanceSettingsWindow

declaration :
bool HGrabModule::ReSizeAdvanceSettingsWindow(int x,int y,int width, int height)

bool Grabber::ReSizeAdvanceSettingsWindow(int x, int y, int width, int height)

description :
Resize advance Settings window size. it's availd after call CreateAdvanceSettingsWindow().

parameters :
x [in] x offset of the new position;
y [in] y offset of the new position;
width [in] Width of the new position;
height [in] Height of the new position;

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ShowAdvanceSettingsWindow

declaration :
bool HGrabModule::ShowAdvanceSettingsWindow(bool show)

bool Grabber::ShowAdvanceSettingsWindow(bool show)

description :
Show advance Settings window. it's availd after call CreateAdvanceSettingsWindow().

parameters :
Show [in] True if show widow or false if hide window

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

CloseAdvanceSettingsWindow

declaration :
bool HGrabModule::CloseAdvanceSettingsWindow()

bool Grabber::CloseAdvanceSettingsWindow()

description :
Close advance Settings window. it's availd after call CreateAdvanceSettingsWindow().

parameters :
None.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetCategoriesCount

declaration :
bool HGrabModule::GetCategoriesCount(long* count)

bool Grabber::GetCategoriesCount(long% count)

description :
This function retrieves the number of grab categories.

parameters :
count [out] number of grab categories.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call

GetLastError to retrieve the error code.

GetCategoriesCap

declaration :
bool HGrabModule::GetCategoriesCap(long index, long* cat_id, LPTSTR name)

bool Grabber::GetCategoriesCap(long index, long% cat_id, String^% name)

description :
This function retrieves the name and the "category identifier" of each category. This identifier is used by all other functions to refer to a specific category.

parameters :
index [in] The index of the category to retrieve. Value must be between 0 and(*ValueCount* - 1).
cat_id [out] The category identifier.
name [out] The name of the category. Allocate at least "TCHAR name[100];" to allocate enough space to write it in.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetFeatureCount

declaration :
bool HGrabModule::GetFeatureCount(long cat_id, long* count)

bool Grabber::GetFeatureCount(long cat_id, long% count)

description :
This function retrieves the number of Feature for special category.

Parameters :
cat_id [in] The category identifier.
count [out] number of grab feature.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetFeatureCap

declaration :
bool HGrabModule::GetFeatureCap(long index, long cat_id,long* feature_id,LPTSTR name, eAdvanceSettingsFeatureType* type, eAdvanceSettingsLevel* level)

bool Grabber::GetFeatureCap(long index, long cat_id, long% feature_id, String^% name,

HEnums::AdvanceSettingsFeatureType% type, HEnums::AdvanceSettingsLevel% level)

description :
This function retrieves the name,feature type and user level, and the "feature identifier" of each feature. Feature type refer the feature's data type(enumeration,Integer, double, Bool and Command). User Level is brief recommended visibility(Beginner, Expert and Guru) feature identifier is used by all other functions to refer to a specific feature.

parameters :
index [in] The index of the feature to retrieve. Value must be between 0 and(*ValueCount* - 1).
cat_id [in] The category identifier.
feature_id [out] The feature identifier.
name[out] The feature name
type [out] The feature type
level[out] The feature visibility level.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Accessible

declaration :
bool HGrabModule::Accessible(long cat_id,long feature_id,BOOL* access)

bool Grabber::Accessible(long cat_id, long feature_id, bool% access)

description :
Check the feature's current status,read-write or read-only.

parameters :
cat_id [in] The category identifier.
feature_id [in] The feature identifier.
access[out] True if read-write false if read-only

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetEnumFeatureValueCount

declaration :
bool HGrabModule::GetEnumFeatureValueCount(long cat_id,long feature_id,long* count)

bool Grabber::GetEnumFeatureValueCount(long cat_id, long feature_id, long% count)

description :
This function retrieves the number of available choices for enumeration feature .

Parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
count[out] The number of available choices

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetEnumFeatureValueCap

declaration :
bool HGrabModule::GetEnumFeatureValueCap(long index, long cat_id,long feature_id,LPTSTR name,long* value)

bool Grabber::GetEnumFeatureValueCap(long index, long cat_id, long feature_id, String^% name, long% value)

description :
This function retrieves the name and value identifier of a selectable choice.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
name[out] The name of the value (ex: Mono, BGR, YUV). Allocate at least "TCHAR name[100];" to allocate enough space to write it in.
value[out] The number of available choices

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetEnumFeatureValue

declaration :
bool HGrabModule::GetEnumFeatureValue(long cat_id,long feature_id,long* value)

bool Grabber::GetEnumFeatureValue(long cat_id, long feature_id, long% value)

description :
This function retrieves the currently selected choice for a particular value.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The value identifier.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call

GetLastError to retrieve the error code.

SetEnumFeatureValue

declaration :
bool HGrabModule::SetEnumFeatureValue(long cat_id,long feature_id,long value)

bool Grabber::SetEnumFeatureValue(long cat_id, long feature_id, long value)

description :
This function selects a new choice for a given setting.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The value identifier.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetIntegerFeatureCap

declaration :
bool HGrabModule::GetIntegerFeatureCap(long cat_id,long feature_id,long *min,long*max)

bool Grabber::GetIntegerFeatureCap(long cat_id, long feature_id, long% min, long% max)

description :
This function retrieves a integer feature range.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
min [out] The feature min value.
max [out] The feature max value.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetIntegerFeatureValue

declaration :
bool HGrabModule::GetIntegerFeatureValue(long cat_id,long feature_id,long* value)

bool Grabber::GetIntegerFeatureValue(long cat_id, long feature_id, long% value)

description :
This function retrieves the integer feature value.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The current value .

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetIntegerFeatureValue

declaration :
bool HGrabModule::SetIntegerFeatureValue(long cat_id,long feature_id,long value)

bool Grabber::SetIntegerFeatureValue(long cat_id, long feature_id, long value)

description :
This function set value to the integer feature.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The current value .

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetDoubleFeatureCap

declaration :
bool HGrabModule::GetDoubleFeatureCap(long cat_id, long feature_id, double* min, double* max)

bool Grabber::GetDoubleFeatureCap(long cat_id, long feature_id, double% min, double% max)

description :
This function retrieves a double feature range.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
min [out] The feature min value.
max [out] The feature max value.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call

GetLastError to retrieve the error code.

GetDoubleFeatureValue

declaration :
bool HGrabModule::GetDoubleFeatureValue(long cat_id,long feature_id,double* value)

bool Grabber::GetDoubleFeatureValue(long cat_id, long feature_id, double% value)

description :
This function retrieves the double feature value.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The current value .

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetDoubleFeatureValue

declaration :
bool HGrabModule::SetDoubleFeatureValue(long cat_id,long feature_id,double value)

bool Grabber::SetDoubleFeatureValue(long cat_id, long feature_id, double value)

description :
This function set value to double feature.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The current value .

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetBoolFeatureValue

declaration :
bool HGrabModule::GetBoolFeatureValue(long cat_id,long feature_id,BOOL* value)

bool Grabber::GetBoolFeatureValue(long cat_id, long feature_id, bool% value)

description :

This function retrieves the Bool feature value.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The current value .

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetBoolFeatureValue

declaration :
bool HGrabModule::SetBoolFeatureValue(long cat_id,long feature_id,BOOL value)

bool Grabber::SetBoolFeatureValue(long cat_id, long feature_id, bool value)

description :
This function set value to Bool feature.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.
value [out] The current value .

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ExecuteCommandFeature

declaration :
bool HGrabModule::ExecuteCommandFeature(long cat_id,long feature_id)

bool Grabber::ExecuteCommandFeature(long cat_id, long feature_id)

description :
This function execute a command defined by the command feature.

parameters :
cat_id [in] The category identifier.
feature_id[in] The feature identifier.

return value:
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

HGrabCallbacks

This class allows the user to be notified when specific events occur in the `HGrabModule` by simply making a class inherit from `HGrabCallbacks` and overriding the functions needing to be custom processed. Important : You should avoid calling HGrabModule's functions from the callback function. For instance, *StopStream()* can not be called in the *OnImageReceived()* function.

For example :

```
CMyClass : public HGrabCallbacks
{
private:
        //custom code...

public:
        virtual void OnImageReceived(HImage* image, unsigned int userData=0,
                                     void* userPtr=0)
        {
                //An image was just received. (stored in the “image” parameter)
                //Now would be a good time to do some processing
                //or maybe save it to disk ?
                .......
        };

        //more custom code...
};
```

Then use the *HGrabModule*'s *SetCallbacks* function to set this class as the one to be called instead of the default, inactive, one.

OnBeforeImageReceived

declaration :
void HGrabCallbacks::OnBeforeImageReceived(unsigned int userData=0, void* userPtr=NULL)

description :
OnBeforeImageReceived is called before the API starts waiting for a new image. Once this callback is done, the grabbing process will enter a waiting state until an image is received. *OnImageReceived* will then be called.

parameters :
userData [out] The value set in the *SetCallback* function.

userPtr [out] The value set in the *SetCallback* function.

return value :
None.

OnImageReceived

declaration :
void HGrabCallbacks::OnImageReceived(HImage* image, unsigned int userData=0, void* userPtr=0)

description :
This function is called when an image has just been received from the grabber.

parameters :
image [out] The image just received from the grabber, converted in HImage format. The received HImage must not be released, as the buffer it comprises will be reused for later grabs.

userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

OnImageReleased

declaration :
void HGrabCallbacks::OnImageReleased(unsigned int userData=0, void* userPtr=0)

description :
This function is called immediately after a captured image buffer is released, making it available again for the GrabModule.

parameters :
userData [out] The value set in the SetCallback function.

userPtr [out] The value set in the SetCallback function.

return value :
None.

OnBeforeQueuingGrab

declaration :
void HGrabCallbacks::OnBeforeQueuingGrab(unsigned int userData=0, void* userPtr=0)

description :
This function is called immediately before a buffer is queued for grabbing.

Parameters :
userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

OnAfterQueuingGrab

declaration :
void HGrabCallbacks::OnAfterQueuingGrab(unsigned int userData=0, void* userPtr=0)

description :
This function is called after a buffer has been successfully queued for grabbing.

parameters :
userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

OnBeforeStartStreaming

declaration :
void HGrabCallbacks::OnBeforeStartStreaming(unsigned int userData=0, void* userPtr=0)

description :
This function is called immediately before starting the streaming process.

Parameters :
userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

OnAfterStartStreaming

declaration :
void HGrabCallbacks::OnAfterStartStreaming(unsigned int userData=0, void* userPtr=0)

description :
This function is called immediately after the streaming process is started.

Parameters :
userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

OnBeforeStopStreaming

declaration :
void HGrabCallbacks::OnBeforeStopStreaming(unsigned int userData=0, void* userPtr=0)

description :
This function is called immediately before stopping the streaming process.

Parameters :
userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

OnAfterStopStreaming

declaration :
void HGrabCallbacks::OnAfterStopStreaming(unsigned int userData=0, void* userPtr=0)

description :
This function is called immediately after the streaming process is halted.

Parameters :
userData [out] The *userData* value set in the *SetCallback* function.

userPtr [out] The *userPtr* value set in the *SetCallback* function.

return value :
None.

HImageExporter

Image Exporter

This module takes one or more images in HImage format and saves them in stand-alone files on disk. Currently supported formats are BMP, JPEG, TIFF and PNG. The following files will need to be placed in the same folder as the application's *exe* : FreeImage.*dll*, libtiff.*dll*, libjpeg.*dll* and zlib.*dll*

Typical usage :

```
HImageExporter exporter;
exporter.BeginExportBmp(_T("c:\\folder\\image"), 1);
exporter.ExportImage(image);
exporter.ExportImage(image2); //(option enabling to export more)
exporter.EndExport();
```

BeginExportBmp

declaration :
bool HImageExporter::BeginExportBmp(LPCTSTR filename, int digits=1)

bool ImageExporter::BeginExportBmp(string fileName, int digits)

description :
BeginExportBmp will prepare the module to successively save images in BMP format. Use in conjunction with ExportImage() and EndExport().

parameters :
filename [in] The full path name of the destination file. (Ex: "C:\folder\image.bmp")

digits [in] The number of digits to append at the end of the filename. For example, if *digits* is 3, files will be exported as : *filename001.bmp, filename002.bmp, filename003.bmp, filename004.bmp,* etc...

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportJpeg

declaration :
bool HImageExporter::BeginExportJpeg(LPCTSTR filename, int digits=1)

bool ImageExporter::BeginExportJpeg(string fileName, int digits)

description :
BeginExportJpeg will prepare the module to successively save images in JPEG format. Use in conjunction with ExportImage() and EndExport().

parameters :
filename [in] The full path name of the destination file. (Ex: “C:\folder\image.jpg”)

digits [in] The number of digits to append at the end of the filename. For example, if *digits* is 3, files will be exported as : *filename001.jpg, filename002.jpg, filename003.jpg, filename004.jpg,* etc...

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportTIFF

declaration :
bool HImageExporter::BeginExportTIFF(LPCTSTR filename, bool multiTIFF=false, int digits=1)

bool ImageExporter::BeginExportTiff(string fileName, bool multiTIFF, int digits)

description :
BeginExportTIFF will prepare the module to successively save images in TIFF format. Use in conjunction with ExportImage() and EndExport().

parameters :

filename [in] The full path name of the destination file. (Ex: “C:\folder\image.tif”)

multiTIFF [in] If true, successive images will be saved in a single multipaged TIFF file. If false, each image will be saved in its own TIFF file.

digits [in] The number of digits to append at the end of the filename. For example, if *digits* is 3, files will be exported as : *filename001.tif, filename002.tif, filename003.tif, filename004.tif,* etc...

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportPng

declaration :
bool HImageExporter::BeginExportPng(LPCTSTR filename, int digits=1)

bool ImageExporter::BeginExportPng(string fileName, int digits)

description :
BeginExportPng will prepare the module to successively save images in PNG format. Use in conjunction with ExportImage() and EndExport().

parameters :

filename [in] The full path name of the destination file. (Ex: “C:\folder\image.png”)

digits [in] The number of digits to append at the end of the filename. For example, if *digits* is 3, files will be exported as : *filename001.png, filename002.png, filename003.png, filename004.png,* etc...

BeginExportJpeg2k

declaration :
bool BeginExportJpeg2k(LPCTSTR filename, int digits = 1)

bool ImageExporter::BeginExportJpeg2k(string fileName, int digits)

description :
BeginExportJpeg2k will prepare the module to successively save images in JPEG2000 format. To be used in conjunction with ExportImage() and EndExport().

parameters :
filename [in] The full path name of the destination file. (Ex: “C:\folder\image.jp2”)

digits [in] The number of digits to append at the end of the filename. For example, if *digits* is 3, files will be exported as : *filename001.jp2, filename002.jp2, filename003.jp2 , filename004.jp2 ,* etc...

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportFit

declaration :
bool BeginExportFit(LPCTSTR filename, bool multiFit=false,bool exportColor = false,int digits=1)

bool ImageExporter::BeginExportFit(string fileName, bool multiFit, bool exportColor, int digits)

description :
BeginExportFit will prepare the module to successively save images in Fit format. Use in conjunction with ExportImage() and EndExport().

parameters :
filename [in] The full path name of the destination file. (Ex: “C:\folder\image.fit”)

multiFit [in] If true, successive images will be saved in a single multipaged Fit file. If false, each image will be saved in its own Fit file.

exportColor [in] If true, successive images will be divided into 3 individual data files: red, green and blue. The color plan is appended to the file name.

digits [in] The number of digits to append at the end of the filename. For example, if *digits* is 3, files will be exported as : *filename001.fit, filename002.fit, filename003.fit , filename004.fit ,* etc...

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ExportImage

declaration :
bool HImageExporter::ExportImage(HImage* image, LPTSTR exportedFileName = NULL)

bool ImageExporter::ExportImage(HermesImage image, ref string exportedFileName)

description :
ExportImage exports an image to a preselected format. The format is selected by calling a "BeginExport..." function.

parameters :
image [in] The image to be exported (must be a valid HImage).

exportedFileName [out] The name of the image file written on disk. (if NULL, name is not returned)

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :

```
HImageExporter exporter;
char filename[MAX_PATH];
exporter.BeginExportBMP("C:\\image.bmp", 1);
exporter.ExportImage(image); //creates image1.bmp
exporter.ExportImage(image2); //creates image2.bmp
exporter.ExportImage(image3); //creates image3.bmp
exporter.EndExport();
```

EndExport

declaration :
bool HImageExporter::EndExport()

bool ImageExporter::EndExport()

description :
EndExport completes the export procedure and executes to some clean up. Use in conjunction with the BeginExportXXX functions.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportBmpTo

declaration :
bool HImageExporter::BeginExportBmpTo()

bool ImageExporter::BeginExportBmpTo()

description :
BeginExportBmpTo will prepare the module to save images successively in the BMP format. Use in conjunction with ExportImageTo() and EndExportTo().

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportJpegTo

declaration :
bool HImageExporter::BeginExportJpegTo()

bool ImageExporter::BeginExportJpegTo()

description :
BeginExportJpegTo will prepare the module to successively save images in JPEG format. Use in conjunction with ExportImageTo() and EndExportTo().

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportTiffTo

declaration :
bool HImageExporter::BeginExportTiffTo()

bool ImageExporter::BeginExportTiffTo()

description :
BeginExportTIFFTo will prepare the module to successively save images in TIFF format. Use in conjunction with ExportImageTo() and EndExportTo(). For multi-paged tiff files use BeginExportTiff().

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportPngTo

declaration :
bool HImageExporter::BeginExportPngTo()

bool ImageExporter::BeginExportPngTo()

description :
BeginExportPngTo will prepare the module to successively save images in PNG format. Use in conjunction with ExportImageTo() and EndExportTo().

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportJpeg2kTo

declaration :
bool HImageExporter::BeginExportJpeg2kTo()

bool ImageExporter::BeginExportJpeg2kTo()

description :
BeginExportJpeg2kTo will prepare the module to successively save images in Jpeg2000 format. Use in conjunction with ExportImageTo() and EndExportTo().

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

BeginExportFitTo

declaration :
bool HImageExporter::BeginExportFitTo(bool exportColor =false)

bool ImageExporter::BeginExportFitTo(bool exportColor)

description :
BeginExportFitTo will prepare the module to successively save images in Fit format. Use in conjunction with ExportImageTo() and EndExportTo().

parameters :
exportColor [in] If true, successive images will be divided into 3 individual data files: red, green and blue. The color plan is appended to the file name.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ExportImageTo

declaration :
bool HImageExporter::ExportImageTo(HImage* image, LPCTSTR fileName)

bool ImageExporter::ExportImageTo(HermesImage image, string fileName)

description :
ExportImageTo exports an image to a preselected format. The format is selected by calling a “BeginExport...To” function.

parameters :
image [in] The image to be exported (must be a valid HImage).

exportedFileName [in] The name of the image file to create on disk.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :

```
HImageExporter exporter;
char filename[MAX_PATH];
exporter.BeginExportBmpTo();
exporter.ExportImageTo(image, "C:\\image.bmp");
exporter.ExportImageTo(image2, "C:\\other-image.bmp");
exporter.EndExportTo();
```

EndExportTo

declaration :
bool HImageExporter::EndExportTo()

bool ImageExporter::EndExportTo()

description :
EndExportTo completes the export procedure and executes some cleanup. Use in conjunction with the BeginExportXXXTo functions.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetJPEGQuality

declaration :
bool HImageExporter::SetJPEGQuality(eHJpegQuality quality)

bool ImageExporter::SetJPEGQuality(HEnums.JpegQuality quality)

description :
This function allows to change the compression quality format used for images exported to JPEG format. It will not affect any format other than JPEG.

parameters :
quality [in] The JPEG quality used, which will regulate the resulting file size, image quality and compression speed will . Experimenting with the different compression schemes will allow to find the most appropriate method for the user's needs. The possible values are listed below :

H_JPEG_DEFAULT
H_JPEG_FAST
H_JPEG_ACCURATE
H_JPEG_QUALITYSUPERB
H_JPEG_QUALITYGOOD
H_JPEG_QUALITYNORMAL
H_JPEG_QUALITYAVERAGE
H_JPEG_QUALITYBAD

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetJPEGQuality

declaration :
eHJpegQuality HImageExporter::GetJPEGQuality()

HEnums.JpegQuality ImageExporter::GetJPEGQuality()

description :
GetJPEGQuality retrieves the compression quality format used for images exported to JPEG format.

parameters :
None.

return value :
The current "quality" used in JPEG compression. See *SetJPEGQuality* for a list of possible values.

SetJPEGLib

declaration :
bool HImageExporter::SetJPEGLib(bool useIPP)

bool ImageExporter::SetJPEGLib(bool useIPP)

description :
This function tells Hermes which library to use for the JPEG compression. The bundled version of Intel's IPP library is faster than the FreeImage library but some customers experienced problems with it (such as failed compression on some images, resulting in incomplete images. If you experience similar problems, use FreeImage. Default is to use IPP.

parameters :
useIPP [in] True to use IPP, false to use FreeImage.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetDateTimeFormat

declaration :
eHImageDateTimeFormat HImageExporter::GetDateTimeFormat()

HEnums::ImageDateTimeFormat ImageExporter::GetDateTimeFormat()

description :
GetDateTimeFormat retrieves the format used to save the image's timestamp to the TIFF files DATETIME tag.

parameters :
None.

return value :
The current DATETIME format. See *SetDateTimeFormat* for a list of possible values.

SetDateTimeFormat

declaration :
void HImageExporter::SetDateTimeFormat(eHImageDateTimeFormat format)

void ImageExporter::SetDateTimeFormat(HEnums::ImageDateTimeFormat format)

description :
This function tells Hermes which format to use to save the image's timestamp to the TIFF files DATETIME tage.

parameters :
format [in] The DATETIME format. The possible values are listed below :

H_DATETIME_STANDARD
Standard Time [YYYY:MM:DD HH:MM:SS] : the tiff specification format.

H_DATETIME_FULL
Full Time [YYMMDD HHMMSSmmmuuu] : includes milliseconds and microseconds.

H_DATETIME_TIME_T
Raw Time [time_t.mmmuuu] : includes the standard UNIX 32-bit time_t value and the milliseconds/microseconds.

H_DATETIME_LTC
LTC Time [HH:MM:SS-FF] : The source frames must have been captured with the support of a LTC timing device.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

HAviFile

This module takes one or more images in HImage format and saves them to an AVI file on disk. Any installed compression codec can be used.

The list of codecs installed on the computer can be found in the *Sound and Multimedia properties* by accessing the **Hardware>Video Codecs>Properties** menu. When the *Video Codecs* properties window opens, check the *Properties* tab for the list of available codecs.

This module can also playback any AVI file recorded by it. An AVI file can either be used for recording or for playback (not at the same time).

CreateAviFile

declaration :
bool HAviFile::CreateAviFile(LPCTSTR filePath, double framerate)

bool AviFile::CreateAviFile(String^ filePath, double framerate)

description :
CreateAviFile will create a new AVI file and open it in recording mode.

parameters :
filePath [in] The full path name of the AVI file to create. (Ex: “C:\\folder\\movie.avi”)

framerate [in] The default average frame rate of the output AVI. (in frames/second), etc...

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

OpenAviFile

declaration :
bool HAviFile::OpenAviFile(LPCTSTR filePath)

bool AviFile::OpenAviFile(String^ filePath)

description :
OpenFile will open an existing AVI file in playback mode.

parameters :
filePath [in] The full path name of the AVI file to open. (Ex: “C:\\folder\\movie.avi”)

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

AddFrame

declaration :
bool HAviFile::AddFrame(HImage* image)

bool AviFile::AddFrame(HermesImage^ image)

description :
AddFrame add an image at the end of an AVI opened in recording mode.

parameters :
image [in] The image to save.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

CloseAviFile

declaration :
bool HAviFile::CloseAviFile()

bool AviFile::CloseAviFile()

description :
CloseAviFile will close the current AVI file. Call this when you are done with the recording/playback.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StopRecording

declaration :
bool HAviFile::StopRecording()

bool AviFile::StopRecording()

description :
StopRecording will stop the video and audio acquisition of the current AVI file. Call this when you don't have any more video frame to send and you want to stop the audio capture. Once the recording starts (after having called 'AddFrame' for the first time), the audio will be continuously captured until 'StopRecording' or 'CloseAviFile' are called. This does not apply if no audio source

were selected.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetCallbacks

declaration :
bool HAviFile::SetCallbacks(HAviFileCallbacks* userCallbacks, unsigned int userData=0, void* userPtr=NULL)

description :
SetCallbacks sets the callback class used by the AVIFile module. (see the HAviFileCallbacks class)

parameters :
userCallbacks [in] The callback class to use.
userData [in] User specified data that is returned by the callback functions. (optional)
userPtr [in] User specified pointer that is returned by the callback functions. (optional)

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetPlaybackFrameRate

declaration :
int HAviFile::GetPlaybackFrameRate()

double AviFile::GetPlaybackFrameRate()

description :
GetPlaybackFrameRate return the frame rate of the AVI file.

parameters :
None.

return value :
The frame rate.

GetVideoCodecCount

declaration :
int HAviFile::GetVideoCodecCount()

int AviFile::GetVideoCodecCount()

description :
GetVideoCodecCount return the number of available video codecs.

parameters :
None.

return value :
The number of available video codecs.

GetVideoCodecName

declaration :
LPCTSTR HAviFile::GetVideoCodecName(int index)

String^ AviFile::GetVideoCodecName(int index)

description :
GetVideoCodecName will return the name of each video codec.

parameters :
index [in] The index of the video codec for which the name is retrieved. Valid range is [0, GetVideoCodecCount()-1].

return value :
The name of the specified codec.

SetVideoCodec

declaration :
bool HAviFile::SetVideoCodec(int index)

bool AviFile::SetVideoCodec(int index)

description :
SetVideoCodec will select the video codec that will be used for compressing the images.

parameters :
index [in] The index of the video codec to use.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ShowVideoCodecSettings

declaration :
bool HAviFile::ShowVideoCodecSettings()

bool AviFile::ShowVideoCodecSettings()

description :
ShowVideoCodecSettings will display a 3rd party dialog allowing to change the currently selected video codec settings. The availability of this dialog varies from codec to codec.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAudioSourceCount

declaration :
int HAviFile::GetAudioSourceCount()

int AviFile::GetAudioSourceCount()

description :
GetAudioSourceCount return the number of available audio sources.

parameters :
None.

return value :
The number of available audio sources.

GetAudioSourceName

declaration :
LPCTSTR HAviFile::GetAudioSourceName(int index)

String^ AviFile::GetAudioSourceName(int index)

description :
GetAudioSourceName will return the name of each audio source.

parameters :
index [in] The index of the audio source for which the name is retrieved. Valid range is [0, GetAudioSourceCount()-1].

return value :

The name of the specified source.

SetAudioSource

declaration :
bool HAviFile::SetAudioSource(int index)

bool AviFile::SetAudioSource(int index)

description :
SetAudioSource will select the audio source that will be used for audio capture.

parameters :
index [in] The index of the audio source to use.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAudioCodecCount

declaration :
int HAviFile::GetAudioCodecCount()

int AviFile::GetAudioCodecCount()

description :
GetAudioCodecCount return the number of available audio codecs.

parameters :
None.

return value :
The number of available audio codecs..

GetAudioCodecName

declaration :
LPCTSTR HAviFile::GetAudioCodecName(int index)

String^ AviFile::GetAudioCodecName(int index)

description :
GetAudioCodecName will return the name of each audio codec.

parameters :
index [in] The index of the audio codec for which the name is retrieved. Valid range is [0,

GetAudioCodecCount()-1].

return value :
The name of the specified codec.

SetAudioCodec

declaration :
bool HAviFile::SetAudioCodec(int index)

bool AviFile::SetAudioCodec(int index)

description :
SetAudioCodec will select the audio codec that will be used for compressing the audio.

parameters :
index [in] The index of the audio codec to use.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ShowAudioCodecSettings

declaration :
bool HAviFile::ShowAudioCodecSettings()

bool AviFile::ShowAudioCodecSettings()

description :
ShowAudioCodecSettings will display a 3rd party dialog allowing to change the currently selected audio codec settings. The availability of this dialog varies from codec to codec.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ForceFrameRate

declaration :
bool HAviFile::ForceFrameRate(double framerate)

bool AviFile::ForceFrameRate(double framerate)

description :
ForceFrameRate will ignore the rate at which images are capture and will force a specified framerate to the file header..

parameters :
framerate [in] The desired framerate.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsLiveRecording

declaration :
bool HAviFile::IsLiveRecording()

bool AviFile::IsLiveRecording()

description :
IsLiveRecording will return the current true if the live recording mode was set.

parameters :
None.

return value :
True if the live recording is enabled, *false* otherwise.

SetLiveRecording

declaration :
void HAviFile::SetLiveRecording(bool enable)

void AviFile::SetLiveRecording(bool enable)

description :
GetAviMode will enable or disable the live recording mode. Unless the live recording mode is enabled, the AVI file framerate will be the one provided to the "CreateAviFile" function. If the live recording mode is enabled, the average frame rate will be calculated when the recording ends. The formula is the following :

framerate = ('StopRecording()' time - First 'AddFrame()' time) / Number of frames captured.

So the length of the AVI file is read as the time lapse between the time at which the first frame was received via the "AddFrame" function and the time at which StopRecording was called.

parameters :
enable [in] True enable the mode, false disable it.

return value :
None.

GetAviMode

declaration :
eHMovieMode HAviFile::GetAviMode()

HEnums::MovieMode AviFile::GetAviMode()

description :
GetAviMode will return the current AVI mode. This is useful as some functions can only be called when the AVI file is in recording mode while some other can only be called while in playback mode.

parameters :
None.

return value :
The current mode. The possible modes are enumerated in HEnums.h

---

GetFrameCount

declaration :
unsigned long HAviFile::GetFrameCount()

unsigned long AviFile::GetFrameCount()

description :
GetFrameMode can be used to retrieve the number of images in the AVI file.

parameters :
None.

return value :
The frame count..

GetCurrentPosition

declaration :
unsigned long HAviFile::GetCurrentPosition()

unsigned long AviFile::GetCurrentPosition()

description :
GetCurrentPosition will return the index of the current frame while in playback mode.

parameters :

None.

return value :
The current index.

DetectVideoCodecType

declaration :
eHVideoCodec HAviFile::DetectVideoCodecType(int index)

VideoCodec AviFile::DetectVideoCodecType(index)

description :
DetectVideoCodecType can be used to detect if a specific video codec can take additionnal parameters.

parameters :
index [in] The index of the video codec to check.

return value :
The video codec type (enumerated in HEnums.h). If the return value is H_CODEC_UNKNOWN, the codec doesn't offer additionnal parameters.

ResetCodecParams

declaration :
void HAviFile::ResetCodecParams()

void AviFile::ResetCodecParams()

description :
ResetCodecParams will reset the parameters used for codecs enumerated in eHVideoCodec.

parameters :
None.

return value :
None.

SetCineFormCodecRawParams

declaration :
bool HAviFile::SetCineFormCodecRawParams(CCineFormRawParam* params)

bool AviFile::SetCineFormCodecRawParams(CCineFormRawParameters* params)

description :

SetCineFormCodecRawParams will set the additionnal parameters if the video codec is of type H_CODEC_CINEFORM_RAM.

parameters :
params [in] The additionnal parameters.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetCineFormCodecHDParams

declaration :
bool HAviFile::SetCineFormCodecHDParams(CCineFormHDParam* params)

bool AviFile::SetCineFormCodecHDParams(CCineFormHDParameters* params)

description :
SetCineFormCodecHDParams will set the additionnal parameters is the video codec is of type H_CODEC_CINEFORM_HD.

parameters :
params [in] The additionnal parameters.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Play

declaration :
bool HAviFile::Play()

bool AviFile::Play()

description :
Play will start the playback of an AVI file. Only available while in playback mode.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Stop

declaration :
bool HAviFile::Stop()

bool AviFile::Stop()

description :
Stop will stop the playback of an AVI file. Only available while in playback mode.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Pause

declaration :
bool HAviFile::Pause()

bool AviFile::Pause()

description :
Pause will pause/resume the playback of an AVI file. Only available while in playback mode.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsPlaying

declaration :
bool HAviFile::IsPlaying()

bool AviFile::IsPlaying()

description :
IsPlaying tells if the AVI is currently playing. Only available while in playback mode.

parameters :
None.

return value :
True if the AVI is playing, *false* in all other cases (paused, stopped, etc)

IsPaused

declaration :
bool HAviFile::IsPaused()

bool AviFile::IsPaused()

description :
IsPaused tells if the AVI has been paused.

parameters :
None.

return value :
True if paused, *false* otherwise.

IsStopped

declaration :
bool HAviFile::IsStopped()

bool AviFile::IsStopped()

description :
IsStopped tells if the AVI is stopped.

parameters :
None.

return value :
True if stopped, *false* otherwise.

ReadImage

declaration :
bool HAviFile::ReadImage(HImage* image, unsigned int index)

bool AviFile::ReadImage(HermesImage^ image, unsigned int index)

description :
ReadImage will retrieve the image at index. Only available while in playback mode.

parameters :
image [in, out] The image read from the AVI. It is the caller responsability to free the image when done with it.

index [in] The index of the image to read. Valid range is [0, GetFrameCount()-1].

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetIndexAt

declaration :

bool HAviFile::GetIndexAt(long time, unsigned short timeMS, unsigned short timeUS, unsigned int& index)

bool AviFile::GetIndexAt(long time, unsigned short timeMS, unsigned short timeUS, unsigned int% index)

description :
Retrieve the index at the specified time stamp.

parameters :
time [in] The time in time_t format.

timeMS [in] The milliseconds part.

timeUS [in] The microseconds part.

index [out] The frame index.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetLooping

declaration :
void HAviFile::SetLooping(bool enable)

void AviFile::SetLooping(bool enable)

description :
Enable or disable looping while doing playback.

parameters :
enable [in] *True* to loop at the beginning of the AVI when its end is reached, *False* to stop.

return value :
None.

IsLooping

declaration :
bool HAviFile::IsLooping()

bool AviFile::IsLooping()

description :
IsLooping tells if the AVI loops when at the end of playback.

parameters :
None.

return value :
True if looping, *false* otherwise.

GetPlaybackFormat

declaration :
eHAviPlaybackFormat HAviFile::GetPlaybackFormat()

HEnums::AviPlaybackFormat AviFile::GetPlaybackFormat()

description :
Retrieve the current image format used for playback. See HAviFile::SetPlaybackFormat().

parameters :
None.

return value :
The playback format.

SetPlaybackFormat

declaration :
void HAviFile::SetPlaybackFormat(eHAviPlaybackFormat format)

void AviFile::SetPlaybackFormat(HEnums::AviPlaybackFormat format)

description :
Set the image format used during playback. By default, the native format is used but calling this function can force the image to BGR or BGRx.

parameters :
format [in] The format to use (see HEnums.h).

return value :
None.

GetPlaybackRate

declaration :
double HAviFile::GetPlaybackRate()

double AviFile::GetPlaybackRate()

description :
Retrieve the current AVI playback rate in percentage of the original. 1.0 means regular speed, 0.5 is half-speed, 2.0 will play twice as fast and so on.

parameters :
None.

return value :
The playback rate.

SetPlaybackRate

declaration :
void HAviFile::SetPlaybackRate(double rate)

void AviFile::SetPlaybackRate(double rate)

description :
Set the AVI playback rate. Default is 1.0, 0.5 is half-speed, 2.0 will play twice as fast and so on. If you change the playback rate during playback, the HAviFile will automaticaly pause the current playback, set the new rate and then resume the playback.

parameters :
rate [in] The playback rate in percentage.

return value :
None.

IsLiveRecording

declaration :
bool HAviFile::IsLiveRecording()

bool AviFile::IsLiveRecording()

description :
Check if the AVI is being recorded in "live" mode.

parameters :
None.

return value :
True if the AVI recording is done "Live", false otherwise.

SetLiveRecording

declaration :
void HAviFile::SetLiveRecording(bool enable)

void AviFile::SetLiveRecording(bool enable)

description :
SetLiveRecording toggle the recording mode for AVI. If *false*, the AVI file will use the framerate specified in the CreateAviFile function. If *true*, the 'framerate' value be recalculated as follow :
framerate = (Time of 'StopRecording' - Time of the first 'AddFrame') / Number of frames captured.

parameters :
enable [in] True to enable live recording. Live Recording is disabled by default.

return value :
None.

.NET Callback Mechanism (.NET only)

The 2 following delegates are used to create the callback functions.

public delegate void OnAviImageNotificationDelegate(HermesImage^ image, unsigned int index, int userData, IntPtr userPtr);

public delegate void OnNotificationDelegate(int userData, IntPtr userPtr);

SetCallbackData

declaration :
void AviFile::SetCallbackData(unsigned int data)

description :
This function set the user data member than will be returned as a parameter of the callback functions.

parameters :
userData [in] An unsigned integer.

return value :
None.

SetCallbackPtr

declaration :
void AviFile::SetCallbackPtr(IntPtr ptr)

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
userPtr [in] A pointer.

return value :
None.

SetCallbackOnAviImageRead

declaration :
void AviFile::SetCallbackOnAviImageRead(OnAviImageNotificationDelegate^ func)

description :
This function set the callback function called for each image read from the AVI file during playback.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnAviEndPlayback

declaration :
void AviFile::SetCallbackOnAviEndPlayback(OnNotificationDelegate^ func)

description :
This function set the callback function called when the end of the AVI file is reached during playback.

parameters :
func [in] The callback function.

return value :
None.

Cineform Encoder Parameters Definition(.NET only)

CColorMatrix

Description :
A help class that define a color 4x3 correction matrix.

CWhiteBalance

Description :
A help class that define a 4 x1 color white balance matrix.

CCineFormParameter

Description :
The base Cineform color processing parameters

EncoderQuality: Encoder quality
0: Low Quality
1: Medium Quality
3: High Quality
4: FilmScan1
5: Filmscan2
ColorMatrx: 4x3 Color correction matrix
Buffer: Buffer size for Encode, 2 is suggested.
CineFormCurve: The curve type for CineForm
Numerator: Numerator parameters if CurveType is H_CURVE_LINE_CUSTOM
Denominator: Denominator parameters if CurveType is H_CURVE_LINE_CUSTOM

CCineFormRawParameter

Description :
Define a Cinform RAW encoder color processing parameters.
BayerPattern: Bayer pattern
0: GBRG
1: GRBG
2: BGGR
3: RGGB

CCineFormHDParameter

Description :
Define a Cinform RAW encoder color processing parameters.

ForceTo444: Force 444 sampling

HSequence

The HSequence manages sequence files by saving one or more images, in *HImage* format, to a Sequence file on disk or to RAM.

Typical usage :

```
//Saving captured images to a sequence file on disk
HSequence sequence;
sequence.Create(_T("c:\\seqfile.seq"));
sequence.Write(imageA);
sequence.Write(imageB);
sequence.Write(imageC);
sequence.Write(imageD);
sequence.Write(imageE);
//and so on...
sequence.Close();
```

Create

declaration :
bool HSequence::Create(LPCTSTR seqPath=NULL, eHCompression compression=H_COMPRESSION_NONE, unsigned long fixedSizeInByte = 0,int quality=100)

bool Sequence::Create(string seqPath, HEnums.Compression compression, uint fixedSizeInByte, int quality)

description :
Creates an empty sequence file on disk or in RAM, resetting the following settings to their default values : looping, playback speed, playback frame skip, playback range.

parameters :
seqPath [in] The full path where the sequence is to be saved (*ex: c:\capture.seq*). Any sequence with the same name will be overwritten. If seqPath is empty, the sequence is created in RAM.

compression [in] The compression used to store images in the sequence. Default is raw uncompressed images. See other formats supported in *HEnums.h.*

fixedSizeInByte [in] Is used to force a constant image size for compressed sequences. The default value, 0, means that no fixed size is imposed. Warning : Forcing a fixed size will truncate images which are larger than fixedSize.

quality [in] The format quality settings (used by the *H_COMPRESSION_JPEG*). An higher quality incurs less image degradation but less compression.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Open

declaration :
bool HSequence::Open(LPCTSTR seqPath)

bool Sequence::Open(string seqPath)

description :
Opens an existing sequence file on disk, automatically detecting the sequence format and opening it accordingly. Opening a sequence will reset the following settings to their default values : looping, playback speed, playback frame skip, playback range.

parameters :
seqPath [in] The full path where the sequence is located. (ex : c:\capture.seq).

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Close

declaration :
bool HSequence::Close()

bool Sequence::Close()

description :
Closes an open sequence, writing the sequence header to the sequence file and closing it. RAM sequence are permanently lost upon closure.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Read

declaration :
bool HSequence::Read(HImage* image, unsigned int index)

bool Sequence::Read(HermesImage image, uint index)

description :
This will read an image from the current sequence (on disk or in RAM). An *Himage* will need to be allocated to be read and freed once it is done with. Refer to the example below.

parameters :
image [out] An *HImage* to hold the image data read from the sequence.

index [in] The requested image index. This ranges from 1 to the value retrieved with *GetAllocatedFrames*.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :

```
HSequence sequence;

//Creates or opens a sequence file.  The sequence must not be empty.
...

//Allocate a HImage to hold the image that will be read.
HImage* image = HImage::CreateHImage();

//Read the first image of the sequence
sequence.Read(image, 1);

//Display the image or processes it
...

//Deallocate the HImage
HImage::DeleteHImage(image);
```

Write

declaration :
bool HSequence::Write(HImage* image)

bool Sequence::Write(HermesImage image)

description :
Writes the specified *image* in the current sequence, which will be appended to the end of the sequence. However, using *SetRecordingPosition* will set a specific recording position, overwriting previously recorded images. Once the function completes, the recording position will be set to append the next image to the current one.

parameters :
image [in] The image to be saved to the sequence.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call

GetLastError to retrieve the error code.

EnableMetadata

declaration :
bool EnableMetadata(unsigned int metadataSize=0, unsigned int metadataInfoSize=0)

description :
Tell the sequence to any metadata bundled with the frames. To enable metadata acquisition, call this function after creating the sequence (but BEFORE the first image is written), i.e. between calls to 'Create' and 'Write'.

parameters :
metadataSize [in] The maximal size taken by the metadata of each image. If '0', the size will be allocated dynamicaly. However, dynamic storage prevent recording in loop.

metadataInfoSize [in] The size of the metadata info section. Only used if you create your own metadata types.

return value :
True if the function is successful or *false* if the function failed.

PreAllocateImages

declaration :
bool HSequence::PreAllocateImages(unsigned int imageSizeBytes, unsigned int count)

description :
Pre-allocate images for use in a RAM sequence. This only works if the current sequence is an uncompressed RAM sequence. Try this is the normal capture to RAM is dropping frames.

parameters :
imageSizeBytes [in] The size in bytes of the images that will be saved to the sequence. You can get this value from the HImage::GetImageSizeBytes().

count [in] The number of image to preallocate. Make sure you don't exceed your current amount of available RAM.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetPreAllocatedImages

declaration :
unsigned int HSequence::GetPreAllocatedImages()

description :
Retrieve the current number of pre-allocated images. As the pre-allocated images are used while recording, this value will goes down.

parameters :
None.

return value :
The number of pre-allocated images.

SetPlaybackPosition

declaration :
void HSequence::SetPlaybackPosition(unsigned int index)

void Sequence::SetPlaybackPosition(uint index)

description :
This function allows to set the start position of the playback.

parameters :
index [in] The desired playback position.

return value :
None.

example :

see HSequence::Play()

GetPlaybackPosition

declaration :
unsigned int HSequence::GetPlaybackPosition()

uint Sequence::GetPlaybackPosition()

description :
Retrieves the current playback position, by returning sequence index of the current playback position.

parameters :
None.

return value :
The current playback position.

SetRecordingPosition

declaration :
void HSequence::SetRecordingPosition(unsigned int index)

void Sequence::SetRecordingPosition(uint index)

description :
SetRecordingPosition specifies where the next image written will be stored in the current sequence. Some sequence formats, such as JPEG, will not allow the overwriting of previous images. SetRecordingPosition will do nothing if that is the case.

parameters :
index [in] The requested recording position.

return value :
None.

example :

```
//Recording in a 10 images loop

HSequence sequence;
sequence.Create("C:\\sequence.seq");

***

//Call SaveImage when an image is received
void SaveImage(HImage* image)
{
	//Write the image to the sequence
	sequence.Write(image);

	//Once 10 images have been captured, continue capture the beginning
	if(sequence.GetRecordingPosition() == 10)
		sequence.SetRecordingPosition(1);
}
```

GetRecordingPosition

declaration :
unsigned int HSequence::GetRecordingPosition()

uint Sequence::GetRecordingPosition()

description :
GetRecordingPosition retrieves the position where the next image will be written in the current

sequence.

parameters :
None.

return value :
The current recording position.

example :
see HSequence::SetRecordingPosition()

SetFrameRate

declaration :
void HSequence::SetFrameRate(double frameRate)

void Sequence::SetFrameRate(double framerate)

description :
Modifies the average frame rate of a sequence. By default, the average frame rate of a sequence is set at 30 fps. However, depending on the capture rate of the application, this rate might need to be adjusted. This is the value written in the sequence file header and is not used anywhere else.

parameters :
frameRate [in] The recommended frame rate in frames per second.

return value :
None.

GetFrameRate

declaration :
double HSequence::GetFrameRate()

double Sequence::GetFrameRate()

description :
GetFrameRate retrieves the average frame rate for the current sequence, in FPS. This is the value written in the sequence file header.

parameters :
None.

return value :
The frame rate of the sequence.

Play

declaration :
bool HSequence::Play()

bool Sequence::Play()

description :
Initiates playback of a sequence from its current position. The current position can be retrieved and set with the *GetPlaybackPosition* and *SetPlaybackPosition* functions before calling *Play*. The playback is done according to the timestamp of each image. Images will be sent by the callbacks functions (see *HSequencerCallbacks*).

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :

```
//Starting a playback from the beginning of the current sequence.
sequence.SetPlaybackPosition(1);
sequence.Play();
```

PlayFixedRate

declaration :
bool HSequence::PlayFixedRate()

bool Sequence::PlayFixedRate()

description :
Initiates playback of a sequence from its current position. The current position can be retrieved and set with the *GetPlaybackPosition* and *SetPlaybackPosition* functions before calling *Play*. Here, the playback proceeds at a fixed speed, independent on the time stamps. The default playback speed is 30 fps, but can be changed to anything using *SetPlaybackSpeed*. Images will be sent by the callbacks functions (see *HSequencerCallbacks*).

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :

```
//Starting a fixed rate playback from the beginning of the current sequence.
```

```
sequence.SetPlaybackPosition(1);
sequence.PlayFixedRate();
```

IsPlayFixedRate

declaration :
bool HSequence::IsPlayFixedRate()

bool Sequence::IsPlayFixedRate()

description :
Use this function to know if the playback is done (or will be done) at a fixed rate. If *Play* was called to start the playback, the function returns false. If *PlayFixedRate* was called to start the playback, the function returns true. If there currently is no playback running (*IsPlaying* returns false), the function will return the last known state.

parameters :
None.

return value :
True the playback proceeds at the frame rate specified by the user. *False* if playback proceeds following the images timestamps.

SetPlaybackSpeed

declaration :
void HSequence::SetPlaybackSpeed(double fps)

void Sequence::SetPlaybackSpeed(double fps)

description :
Sets the playback framerate to be used with *PlayFixedRate,* the default value being 30 fps. This function can also be called at any time during a fixed rate playback to change speed on-the-fly. Specifying a negative value will play the sequence backward. This function has no impact on the *Play* function.

parameters :
fps [in] The playback speed, in frames/seconds. Positive for forwards playback, negative for backwards playback.

return value :
None.

GetPlaybackSpeed

declaration :
double HSequence::GetPlaybackSpeed()

double Sequence::GetPlaybackSpeed()

description :
Retrieves the speed specified in *PlayFixedRate*. See *SetPlaybackSpeed* for additional information.

parameters :
None.

return value :
The current playback speed in frames per second.

SetPlaybackFrameSkip

declaration :
void HSequence::SetPlaybackFrameSkip(unsigned int frameSkip)

void Sequence::SetPlaybackFrameSkip(uint frameSkip)

description :
Sets the frame skip used with *PlayFixedRate*. This function can also be called at any time during a fixed rate playback to change the frame skip on-the-fly. For instance, if the starting playback position is 1 and the *frameSkip* is 1, you will get frames 1, 3, 5, 7, 9, etc. Default value is 0. This function has no impact on the *Play* function.

parameters :
frameSkip [in] The frame skip value.

return value :
None.

GetPlaybackFrameSkip

declaration :
unsigned int HSequence::GetPlaybackFrameSkip()

uint Sequence::GetPlaybackFrameSkip()

description :
Retrieves the frame skip used with *PlayFixedRate*. See *SetPlaybackFrameSkip* for additional information.

parameters :
None.

return value :
The current frame skip value.

SetMaxPlaybackInterval

declaration :
void HSequence::SetMaxPlaybackInterval(int ms)

void Sequence::SetMaxPlaybackInterval(int ms)

description :
When a playback is done using the Play() function, the playback is done following the timestamps. So if two successive images are 10 seconds appart, Hermes will wait 10 seconds between the sending of each image. This can be a problem is the recording is not done continuously. For example, you start a recording, stop it, then restart it later. To work around this problem, Hermes uses a maximum playback interval value. For example, if you set this value to 1000 ms, the playback will proceed with, at most, a one second interval between each successive frames. Default value is 5000 ms.

parameters :
ms [in] The maximum playback interval value in milliseconds.

return value :
None.

GetMaxPlaybackInterval

declaration :
int HSequence::GetMaxPlaybackInterval()

int Sequence::GetMaxPlaybackInterval()

description :
Retrieves the maximum playback interval used by the Play() function.

parameters :
None.

return value :
The current maximum playback interval value in milliseconds.

Stop

declaration :
bool HSequence::Stop()

bool Sequence::Stop()

description :
Stops sequence playback.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetPlaybackRange

declaration :
void HSequence::SetPlaybackRange(unsigned int start, unsigned int end)

void Sequence::SetPlaybackRange(uint start, uint end)

description :
SetPlaybackRange restricts playback to a specific frame range. If playback is started outside of the *start* and *end* bounds, the current playback position will automatically snap to *start*. This only affects the *Play* and *PlayFixedRate* functions. When *end* is reached, if looping is enabled, playback will continue from *start*. Otherwise, playback will stop. To disable ranged playback, set both parameters to 0 (which are the default values).

parameters :
start [in] The first frame to be used in playback.

end [in] The last frame to be used in playback.

return value :
None.

GetPlaybackRange

declaration :
void HSequence::GetPlaybackRange(unsigned int& start, unsigned int& end)

void Sequence::GetPlaybackRange(ref uint start, ref uint end)

description :
GetPlaybackRange returns the current range used by playback functions. If start and end are 0, ranged playback is disabled.

parameters :
start [out] The first frame used in playback.

end [out] The last frame used in playback.

return value :
None.

IsPlaying

declaration :
bool HSequence::IsPlaying()

bool Sequence::IsPlaying()

description :
Retrieves the playback state, *i.e.* playing or not.

parameters :
None.

return value :
True if playback is running, *false* if playback stopped.

example :

```
HSequence sequence;

void TogglePlayback()
{
      if(sequence.IsPlaying())
            sequence.Stop();
      else
            sequence.Play();
}
```

IsRAMSequence

declaration :
bool HSequence::IsRAMSequence()

bool Sequence::IsRAMSequence()

description :
Retrieves the Ram sequence storage state. *I.e.* if the sequence is stored in RAM

parameters :
None.

return value :
True if the sequence is stored in RAM, *false* otherwise.

IsSequenceOpen

declaration :
bool HSequence::IsSequenceOpen()

bool Sequence::IsSequenceOpen()

description :
Check the sequence state.

parameters :
None.

return value :
True if a sequence is currently loaded, *false* otherwise.

GetSequenceCompression

declaration :
eHCompression HSequence::GetSequenceCompression()

HEnums.Compression Sequence::GetSequenceCompression()

description :
GetSequenceCompression retrieves the compression used by the sequence to store images. See *HEnums.h* for the description of each format.

parameters :
None.

return value :
The sequence format.

SupportRewrite

declaration :
bool HSequence::SupportRewrite()

bool Sequence::SupportRewrite()

description :
SupportRewrite verifies if the current sequence supports rewrites. If not, *SetRecordingPosition* will not have any effect and recording will be done sequentially. RAM sequences always support rewrites. Sequences using a compressed format will not support rewrites as the image size is variable, meaning that rewrite could overwrite part of the following frame(s) in the sequence.

parameters :
None.

return value :
True if the sequence supports rewrites.

GetImageWidth

declaration :
unsigned int HSequence::GetImageWidth()

uint Sequence::GetImageWidth()

description :
GetImageWidth retrieves the width of the images stored in the sequence.

parameters :
None.

return value :
The image width in pixels.

GetImageHeight

declaration :
unsigned int HSequence::GetImageHeight()

uint Sequence::GetImageHeight()

description :
GetImageHeight retrieves the height of the images stored in the sequence.

parameters :
None.

return value :
The image height in pixel.

GetImageBitDepth

declaration :
unsigned int HSequence::GetImageBitDepth()

uint Sequence::GetImageBitDepth()

description :
GetImageBitDepth retrieves the bit depth of the images stored in the current sequence.

parameters :
None.

return value :
The image bit depth.

GetImageBitDepthReal

declaration :
unsigned int HSequence::GetImageBitDepthReal()

uint Sequence::GetImageBitDepthReal()

description :
GetImageBitDepthReal retrieves the real bit depth of the images stored in the current sequence.

parameters :
None.

return value :
The image valid bit depth.

GetImageSizeBytes

declaration :
unsigned int HSequence::GetImageSizeBytes()

uint Sequence::GetImageSizeBytes()

description :
GetImageSizeBytes retrieve the size in bytes of each image stored in the sequence.

parameters :
None.

return value :
The image size in bytes.

GetImageFormat

declaration :
eHImageFormat HSequence::GetImageFormat()

HEnums.ImageFormat Sequence::GetImageFormat()

description :
GetImageFormat retrieve the format of the images stored in the sequence.

parameters :
None.

return value :

The image format.

GetAllocatedFrames

declaration :
unsigned int HSequence::GetAllocatedFrames()

uint Sequence::GetAllocatedFrames()

description :
GetAllocatedFrames retrieves the number of images stored in the current sequence.

parameters :
None.

return value :
The number of images in the sequence.

GetTrueImageSize

declaration :
unsigned int HSequence::GetTrueImageSize()

uint Sequence::GetTrueImageSize()

description :
GetTrueImageSize retrieves the size, of each image in the sequence, in bytes. That size can be larger that the actual size of each image, due to alignment boundaries considerations. The space a sequence will take on disk can be estimated by multiplying this value by the number of images in the sequence.

parameters :
None.

return value :
The size of one image on disk.

DumpHeader

declaration :
void HSequence::DumpHeader()

void Sequence::DumpHeader()

description :
Normally, the sequence header is only written to the sequence file when the sequence is closed. However, when capturing long or unsupervised sequences where power failures or system crashes

could happen, *DumpHeader* can be periodically called as a safety net. *DumpHeader* writes the sequence header to the sequence file *immediately,* saving all frames captured until called and allowing later access to an interrupted recording session. This will not have any effect on a RAM sequence.

parameters :
None.

return value :
None.

Truncate

declaration :
void HSequence::Truncate(unsigned int index)

void Sequence::Truncate(uint index)

description :
Use *Truncate* to remove all frames from "index", up to the last frame of the sequence. Frames located before "index" will not be deleted. If index is "1", the entire sequence will be cleared.

parameters :
index [in] The frame index to truncate at.

return value :
None.

GetDescription

declaration :
bool HSequence::GetDescription(BYTE* description)

bool Sequence::GetDescription(ref string description)

description :
GetDescription allows to retrieve the 512 bytes of custom data included in the sequence header. See *SetDescription* for more info.

parameters :
description [out] The description from the sequence header will be written here. Parameter allocation is user selected and could be "BYTE description[512];" .

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetDescription

declaration :
bool HSequence::SetDescription(BYTE* description)

bool Sequence::SetDescription(string description)

description :
The header of a sequence file comprises 512 bytes to be used at the user discretion. They can be used to store a description of the sequence, identifiers, related external data or anything else.

parameters :
description [in] The description to be written to the sequence header. *description* must be allocated to hold at least 512 bytes. The 512 bytes will be copied as-is.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetDescriptionFormat

declaration :
eDescriptionFormat HSequence::GetDescriptionFormat()

description :
Retrieves the format of the description written in the sequence header. See *SetDescriptionFormat* for more info.

parameters :
None.

return value :
The format of the description. (see *SetDescriptionFormat*)

SetDescriptionFormat

declaration :
void HSequence::SetDescriptionFormat(eDescriptionFormat format)

description :
The description in the sequence header can come in one of the following formats. These formats indicate the method used to interpret it:

H_UNICODE_FORMAT : The description is an UNICODE string that can be casted to a (wchar_t*)

H_MBCS_FORMAT : The description is a MBCS string that can be casted to a (char*)

H_BYTE_FORMAT : The description is binary data. Interpretation is left at user discretion.

parameters :
format [out] The format of the description.

return value :
None.

SetCallbacks

declaration :
bool HSequence::SetCallbacks(HSequencerCallbacks* userCallbacks, unsigned int userData=0, void* userPtr=NULL)

description :
SetCallbacks sets the callback class used by the sequencer module. (see the HSequencerCallbacks class)

parameters :
userCallbacks [in] The callbacks class to use. This will be a pointer to one the user's *HSequencerCallbacks*-derived class.

userData [in] This parameter can be used to pass a value to be used in the user's callback functions.

userPtr [in] This parameter can be used to pass a pointer to be used the user's callback functions.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ReadTimestamp

declaration :
bool HSequence::ReadTimestamp(unsigned int index, long& time, unsigned short& timeMS, unsigned short& timeUS)

bool Sequence::ReadTimestamp(uint index, ref int time, ref ushort timeMS, ref ushort timeUS)

description :
ReadTimestamp will retrieve the timestamp of a given frame in the sequence.

parameters :
index [in] The frame index. This value must be between 1 and the frame count.

time [out] The absolute timestamp in seconds.

timeMS [out] The millisecond part of the timestamp.

timeUS [out] The microsecond part of the timestamp.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetIndexAt

declaration :
bool HSequence::GetIndexAt (long time, unsigned short timeMS, unsigned short timeUS, unsigned int& index)

bool Sequence::GetIndexAt(int time, ushort timeMS, ushort timeUS, ref uint index)

description :
GetIndexAt searches the sequence to find the last frame that had been captured at a given time. It returns the index of that frame.

parameters :
time [in] The absolute timestamp in seconds.

timeMS [in] The millisecond part of the timestamp.

timeUS [in] The microsecond part of the timestamp.

index [out] The frame index.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetIndexNear

declaration :
bool HSequence::GetIndexNear(long time, unsigned short timeMS, unsigned short timeUS, unsigned int& index)

bool Sequence::GetIndexNear(int time, ushort timeMS, ushort timeUS, ref uint index)

description :
GetIndexNear searches the sequence to find the frame which is the nearest in time to the given time. It returns the index of that frame.

parameters :
time [in] The absolute timestamp in seconds.

timeMS [in] The millisecond part of the timestamp.

timeUS [in] The microsecond part of the timestamp.

index [out] The frame index.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetReferenceFrame

declaration :
void HSequence::SetReferenceFrame(unsigned long index)

void Sequence::SetReferenceFrame(uint index)

description :
SetReferenceFrame saves the index of the frame to use as the reference. An application can use this value to mark a specific frame in the sequence as the "event" of the sequence. For example, this value could be used to show time stamps relative to this frame. The reference frame is 0 by default (i.e. no reference frame)

parameters :
index [in] The index of the reference frame.

return value :
None.

GetReferenceFrame

declaration :
unsigned int HSequence::GetReferenceFrame()

uint Sequence::GetReferenceFrame()

description :
GetReferenceFrame retrieve the index of the reference frame for the current sequence. If no reference frame was defined, the returned value is 0.

parameters :
None.

return value :
The index of the reference frame.

Sort

declaration :

void HSequence::Sort()

void Sequence::Sort()

description :
Sort will sort each frame of a sequence by timestamp and copy them to an empty sequence. This is useful to save a sequence captured in a recording loop, where frame 1 is not necessarily the oldest frame. Once the copy is done, the current sequence will destroy itself, rename the created sequence file by its own name and automatically open it. Obviously, calling sort while in the middle of a recording or playback is not recommended. *Sort* can be a lengthy operation depending of the number of frames to be sorted and is not available for compressed sequence formats.

parameters :
None.

return value :
None.

ResynchronizePlayback (1)

declaration :
bool HSequence::ResynchronizePlayback(long offset, unsigned short offsetMS)

description :
This function will put an offset on the start time reference of a timestamped sequence playback. Sending a negative offset will make the playback accellerate. For example, a "-3 seconds" offset tells the playback that it is 3 second late and must do some catch-up. A positive offset will make the playback stalls for the specified amount of time. This function only works on timestamped playback (not on manual playback).

parameters :
offset [in] The offset to apply (in seconds).

offsetMS [in] The offset to apply (milliseconds part)

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ResynchronizePlayback (2)

declaration :
bool HSequence::ResynchronizePlayback(unsigned long index)

description :
This function will jump to the frame index provided and continue playback from there. This function only works on timestamped playback (not on manual playback).

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetNextPlaybackFrameTime

declaration :
bool HSequence::GetNextPlaybackFrameTime(long& time, unsigned short& timeMS, unsigned short& timeUS)

description :
This function returns the time stamp of the next image that will be shown during a playback.

parameters :
time [out] The time stamp of the next frame.

timeMS [out] The time stamp of the next frame (milliseconds).

timeUS [out] The time stamp of the next frame (microseconds).

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsBayer

declaration :
bool HSequence::IsBayer()

bool Sequence::IsBayer()

description :
IsBayer tests if the sequence has one of the following image format:

H_IMAGE_MONO_BAYER,
H_IMAGE_MONO_BAYER_JPEG,
H_IMAGE_MONO_BAYER_RLE,
H_IMAGE_MONO_BAYER_HUFFMAN,
H_IMAGE_MONO_BAYER_LZ,
H_IMAGE_MONO_BAYER_RLE_FAST,
H_IMAGE_MONO_BAYER_HUFFMAN_FAST,
H_IMAGE_MONO_BAYER_LZ_FAST,
H_IMAGE_MONO_BAYER_MSB,
H_IMAGE_MONO_BAYER_MSB_SWAP,
H_IMAGE_MONO_BAYER_PPACKED,
H_IMAGE_MONO_BAYER_PPACKED_8448.

parameters :
None.

return value :
True if a Bayer conversion can be applied on the current sequence, *false* otherwise.

GetBayerPattern

declaration :
bool HSequence::GetBayerPattern(eHBayerPatternOrigin& pattern)

bool Sequence::GetBayerPattern(ref HEnums.BayerPatternOrigin pattern)

description :
GetBayerPattern retrieves the pattern origin, from the sequence header. If no pattern was set, the returned value is H_BAYER_PATTERNORIGIN_GB.

parameters :
pattern [out] The pattern origin. See HEnums.h for all possible patterns.

return value :
True if the sequence is Bayer, *false* otherwise. See IsBayer method for more details.

SetBayerPattern

declaration :
void HSequence::SetBayerPattern(eHBayerPatternOrigin pattern)

void Sequence::SetBayerPattern(HEnums.BayerPatternOrigin pattern)

description :
SetBayerPattern saves the pattern origin, used for a Bayer conversion, into the sequence header. An application can use this value to store the right pattern origin, for later use.

parameters :
pattern [in] The pattern origin. See HEnums.h for all possible patterns.

return value :
None.

.NET Callback Mechanism (.NET only)

The 3 following delegates are used to create the callback functions.

public delegate void OnImageNotificationDelegate(HermesImage image, int userData, IntPtr userPtr);

public delegate void OnLoopNotificationDelegate(bool% loop, int userData, IntPtr userPtr);

public delegate void OnNotificationDelegate(int userData, IntPtr userPtr);

SetCallbackData

declaration :
void Sequence::SetCallbackData(unsigned int data)

description :
This function set the user data member than will be returned as a parameter of the callback functions.

parameters :
userData [in] An unsigned integer.

return value :
None.

SetCallbackPtr

declaration :
void Sequence::SetCallbackPtr(IntPtr ptr)

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
userPtr [in] A pointer.

return value :
None.

SetCallbackOnImageRead

declaration :
void Sequence::SetCallbackOnImageRead(OnImageNotificationDelegate func)

description :

This function set the callback function called for each image read from the sequence during playback.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnImageWrite

declaration :
void Sequence::SetCallbackOnImageWrite(OnNotificationDelegate func)

description :
This function set the callback function called after an image get written to the current sequence.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnEndOfSequence

declaration :
void Sequence::SetCallbackOnEndOfSequence(OnLoopNotificationDelegate func)

description :
This function set the callback function called when the end of the sequence is reached during playback.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnEndOfPlaybackRange

declaration :
void Sequence::SetCallbackOnEndOfPlaybackRange(OnLoopNotificationDelegate func)

description :
This function set the callback function called when either the beginning or end of the playback range sequence is reached during playback.

parameters :
func [in] The callback function.

return value :
None.

SetCallbackOnPlaybackStopped

declaration :
void Sequence::SetCallbackOnPlaybackStopped(OnNotificationDelegate func)

description :
This callback function will be called when the playback stops, either by reaching the end of the sequence/range or when the Stop function is called.

parameters :
func [in] The callback function.

return value :
None.

HSequencerCallbacks

This class allows the user to be informed when certain events occurs in the *HSequencer* module, by simply making one of the user's class inherit from *HsequencerCallbacks* and overriding the functions needing to be custom processed. It is recommended to limit the actions implemented in the callbacks to the minimum (*i.e.* Some state queries such as getting the current frame, copying the image or processing it.)

A blocking function should never be called in the callbacks. For instance, , calling *Stop()* to halt a playback from the callbacks function will cause a deadlock. It is recommended to avoid using blocking messaging functions (SendMessage, UpdateData, UpdateWindow, SetWindowText, etc.) as this could lead to deadlock situation if a Stop() is received while the callback is getting processed. Instead, use a non-blocking PostMessage() to do the GUI updating in another function.

For example :

```
CMyClass : public HSequencerCallbacks
{
private:
        //custom code...

public:
        virtual void OnImageRead(HImage* image, unsigned int userData,
   void* userPtr)
        {
                //An image was just read. (stored in the “image” parameter)
                //Now would be a good time to display it to the user
                .......
        };

        //more custom code...
};
```

Then use the *HSequencer's SetCallbacks* function to set this class as the one to be called instead of the inactive default class.

OnImageRead

declaration :
virtual void HSequencerCallbacks::OnImageRead(HImage* image, unsigned int userData, void* userPtr)

description :
This callback function is called when an image was just read from a sequence (during a playback for instance)

parameters :
image [out] The image read from the sequence. The images should always be freed once they are not needed anymore. (*i.e. delete image*)

userData [out] The value set in the *SetCallback* function.

userPtr [out] The value set in the *SetCallback* function.

return value :
None.

OnImageWrite

declaration :
virtual void HSequencerCallbacks::OnImageWrite(unsigned int userData=0, void* userPtr=0)

description :
This callback function is called when an image was just written to a sequence (once a "Write" function completes).

parameters :
userData [out] The value set in the *SetCallback* function.

userPtr [out] The value set in the *SetCallback* function.

return value :
None.

OnEndOfSequence

declaration :
virtual void HSequencerCallbacks::OnEndOfSequence(bool& loop, unsigned int userData, void* userPtr)

description :
This callback function will be called after the last image of the sequence is shown during a playback.

parameters :
userData [out] The value set in the SetCallback function.

userPtr [out] The value set in the SetCallback function.

loop [in] If you set it to true, playback will continue from the begining of the sequence. If false, the playback will stop.

return value :
None.

OnEndOfPlaybackRange

declaration :
virtual void HSequencerCallbacks::OnEndOfPlaybackRange(bool& loop, unsigned int userData, void* userPtr)
description :
This callback function will be called when either the beginning or end of the playback range is reached during playback.

parameters :
userData [out] The value set in the SetCallback function.

userPtr [out] The value set in the SetCallback function.

loop [in] If you set it to true, playback will continue from the begining of the sequence. If false, the playback will stop.

return value :
None.

OnPlaybackStopped

declaration :
virtual void HSequencerCallbacks::OnPlaybackStopped(unsigned int userData, void* userPtr)

description :
This callback function will be called when the playback stops, either by reaching the end of the sequence/range or when the Stop function is called.

parameters :
userData [out] The value set in the SetCallback function.

userPtr [out] The value set in the SetCallback function.

return value :
None.

HViewer

Viewer

This module provides features allowing to display *HImages* on screen.

Typical usage :

```
HViewer viewer;

void CYourWindow::OnPaint()
{
	CPaintDC dc(this); // device context for painting

	//LastImage must be a valid HImage
	Viewer.ShowImage(dc.m_hDC, LastImage);
}
```

ShowImage

declaration :
bool HViewer::ShowImage(HDC deviceContext, HImage* image, int offsetX=0, int offsetY=0)

bool Viewer::ShowImage(System.IntPtr deviceContext, HermesImage image, int offsetX, int offsetY)

description :
Display an HImage on a user-created device context.

parameters :
deviceContext [in] Handle to the destination device context.

image [in] The image to be shown.

offsetX [in] The horizontal offset of the image on the canvas.

offsetY [in] The vertical offset of the image on the canvas.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ShowScaledImage

declaration :
bool HViewer::ShowScaledImage(HDC deviceContext, HImage* image, int top, int left, unsigned int width, unsigned int height)

bool Viewer::ShowScaledImage(System.IntPtr deviceContext, HermesImage image, int top, int left,

uint width, uint height)

description :
Displays an *HImage* on an user-created device context. The image is scaled to the specified size. This consumes more CPU time due to the scaling required.

parameters :
deviceContext [in] Handle to the destination device context.

image [in] The image to be shown.

top [in] Specifies the y-coordinate of the upper-left corner of the destination rectangle.

left [in] Specifies the x-coordinate of the upper-left corner of the destination rectangle.

width [in] Specifies the width of the destination rectangle.

height [in] Specifies the height of the destination rectangle.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ShowImageOnDC

declaration :
bool HViewer::ShowImageOnDC(HDC deviceContext, HImage* image, RECT srcRect, RECT destRect, int zoomX=100, int zoomY=100, bool autoScaleUp=false, bool autoScaleDown=false, bool keepAspectRatio=true, RECT* resultRect=NULL)

bool Viewer::ShowImageOnDC(System.IntPtr deviceContext, HermesImage image, int srcLeft, int srcTop, int srcRight, int srcBottom, int destLeft, int destTop, int destRight, int destBottom, int zoomX, int zoomY, bool autoScaleUp, bool autoScaleDown, bool keepAspectRatio, IntPtr resultRect)

description :
Display an *HImage* on a user-created device context, according to the parameters. This consumes more CPU time if scaling is required.

parameters :
deviceContext [in] Handle to the destination device context.

image [in] The image to be shown.

*src*Rect [in] The part of the source image to display.

destRect [in] Where the drawing take place on the DC.

zoomX [in] Zoom percentage in X (overridden by autoScaleUp/autoScaleDown).

zoomY [in] Zoom percentage in Y (overridden by autoScaleUp/autoScaleDown).

autoScaleUp [in] Scale image to fit destRect (if destRect > srcRect).

autoScaleDown [in] Scale to fit destRect (if destRect < srcRect).

keepAspectRatio [in] Keep image aspect ratio when scaling (ignore zoomY).

resultRect [out] Points to a user preallocated data that retrieves the result image rectangle. If NULL is passed, the parameter is ignored.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetZoom

declaration :
void HViewer::SetZoom(int zoomX, int zoomY)

void Viewer::SetZoom(int zoomX, int zoomY)

description :
Sets the horizontal and vertical scaling used when *ShowImage* is called to display an image. This has no impact on *ShowScaledImage* or *ShowImageOnDC* For example, if *zoomX* and *zoomY* are "200" (200%), the image will be shown two time bigger than its actual size. Showing a scaled image will require more CPU time. It is strongly recommended to leave the zooms at "100" (*i.e.* no scaling) when performance is required (when recording for example)

parameters :
zoomX [in] The horizontal scaling (%)
zoomY [in] The vertical scaling (%)

return value :
None.

GetZoom

declaration :
void HViewer::GetZoom(int* zoomX, int* zoomY)

void Viewer::GetZoom(ref int zoomX, ref int zoomY)

description :
Retrieves the horizontal and vertical scaling used when *ShowImage* is called to display an image.

parameters :
zoomX [out] The horizontal scaling (%)

zoomY [out] The vertical scaling (%)

return value :
None.

HImageConverter

ImageConverter

This module can be used to convert any supported image format to the following image formats : Monochrome 8 bits (most basic monochrome image), Color BGR (most basic color image) and Color BGRx (color format compatible with most Win32 display/bitmap functions).

This can be useful for image processing by allowing the use of a single algorithm for all converted images instead of using different ones for each image format. For example, *HImageExporter* and *HAVIExporter* internally uses those functions to insure compatibility of the images with the exporting libraries.

Unless specified, any source image that is more than 8 bit depth will be converted to an 8 bit or 24 bit color image. The 8 most significant bit are kept. Use color mapping via LUT for better levelling (see below).

Note that raw bayer images when converted to BRG will becomes raw bayer RGB images. Not Bayer interpolation will be performed. Call rather HbayerConverter classes for that purpose.

HimageConverter can also preform image conversion using look up table (anamorphosis) operations. As an example, it can be used for remapping specific pixel values to some other values. This can be applied, for instance, for pixel intensity rescaling, contrast, brightness, gamma or color remapping correction.

(All colorRemapping function is obsolete please using HImageProcessing instead)

To enable color remapping functionality call SetColorRemapping() before calling the convert function. For example, to tranform a source image to a color BRG image using color remapping with the rainbow pseudo colors:

```
ImageConvert-> SetColorRemapping(true);
....
ImageConvert->SetMonoPseudoColorMode(H_MONO_TO_PSEUDO_COLOR_LUT_RAINBOW);
....
ImageConvert->ConvertToBGR( imageIn, imageOut);
```

ConvertToBGR

declaration :
bool HImageConverter::ConvertToBGR(HImage* imageIn, HImage* imageOut, bool flip)

bool ImageConverter::ConvertToBGR(HermesImage imageIn, HermesImage imageOut, bool flip)

description :
Converts an image to the BGR format with 24 bits per pixel (Blue-Green-Red).

parameters :

imageIn [in] The source image.

imageOut [out] Where the converted image is stored. The related target is created with HImage::CreateHImage().

flip [in] If this is *true*, the image will be also be horizontally flipped. It is required for some image libraries but some input formats are not supported yet .

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code. Possible failure may be due to the fact that the source image format is not supported for that conversion.

ConvertToBGRx

declaration :
bool HImageConverter::ConvertToBGRx(HImage* imageIn, HImage* imageOut)

bool ImageConverter::ConvertToBGRx(HermesImage imageIn, HermesImage imageOut)

description :
Converts an image to the BGRx format with 32 bits per pixel (Blue-Green-Red-Filler).

parameters :
imageIn [in] The source image.

imageOut [out] Where the converted image is stored. The related target is created with HImage::CreateHImage().

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code. Possible failure may be due to the fact that the source image format is not supported for that conversion.

ConvertToMono8

declaration :
bool HImageConverter::ConvertToMono8(HImage* imageIn, HImage* imageOut, bool flip)

bool ImageConverter::ConvertToMono8(HermesImage imageIn, HermesImage imageOut, bool flip)

description :
Convert an image to the Mono 8 format. This function will only accept Monochrome images input.

parameters :
imageIn [in] The source image.

imageOut [out] Where the converted image is stored. The related target is created with

HImage::CreateHImage().

flip [in] If this is true, the image will be also be horizontally flipped (upside down). Some image libraries require this.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code. Possible failure may be due to the fact that the source image format is not supported for that conversion.

SetColorRemapping

Declaration :
void HImageConverter::SetColorRemapping(bool enable)

description :
Enable Color Remapping via LUT.

parameters :
enable[IN]: True: enable mapping via LUT. False: disable LUT mapping.
return value :

IsColorRemappingEnabled

declaration :
bool HImageConverter::IsColorRemappingEnabled()

description :
Check if Color remapping feature is on or off.

parameters :

return value :
True if color remapping feature is on, else return false;

SetASCColorRemapping

declaration :
void HImageConverter::SetASCColorRemappingParam(bool enable)

description :
Enable LUT remapping using the American Society of Cinematographers Color Decision List (ASC CDL). This define a format for the exchange of basic primary color grading information. A 3 parameters formula can be used to define most possible remapping and color correction:

<u>The formula for ASC CDL color correction is:</u>

where:

- *out* is the color graded pixel code value
- *i* is the input pixel code value (0=black, 1=white)
- *s* is slope (any number 0 or greater, nominal value is 1.0)
- *o* is offset (any number, nominal value is 0)
- *p* is power (any number greater than 0, nominal value is 1.0)

The formula is applied to the three color values for each pixel using the corresponding slope, offset, and power(gamma) for each color channel.

Parameters
enable:True if ASC color mapping enabled else return false

return value :

IsASCColorRemappingEnabled

declaration :
bool HImageConverter::IsASCColorRemappingEnabled()

description :
Check if ASC color mapping eanbled

parameters :

return value :
True if ASC color mapping enabled else return false;

GetASCColorRemappingParam

declaration :
bool HImageConverter::GetASCColorRemappingParam(double ascSlope[3],double ascOffset[3], double ascPower[3])

description :
Get current ASC parameters settings. Array Index 0 is Blue channel ;1 is Green channel; 2 is Red channel. When processing monochrome image use index 0 only.

parameters :
ascSlope [out]: ASC slope parameter.
ascOffset [out]: ASC offset parameter.
ascPower [out]: ASC power parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetASCColorRemappingParam

declaration :
bool HImageConverter::SetASCColorRemappingParam(double ascSlope[3],double ascOffset[3], double ascPower[3])

description :
Set ASC Parameter. Array Index 0 is Blue channel ;1 is Green channel; 2 is red channel. When processing monochrome image use index 0 only.

parameters :
ascSlope [IN]: ASC slope parameter.
ascOffset [IN]: ASC offset parameter.
ascPower [IN]: ASC power parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetColorClipLevel

declaration :
void HImageConverter::SetColorClipLevel(int level)

description :
This function is only available for pixel depth greater than 8 bits. By default, HImageConvert always return the 8 most significant bits of captured images. This function can highlight specific bit ranges on the images.

As an example, a 10 bit image can be leveled in 3 ways:

- level 2: 8 most significant bits (bit 2 to 10, default HImageConvert setting)
- level 1: 8 middle significant bits (bit 1 to 9)
- level 0: 8 least significant bits (bit 0 to 7)

parameters :
level[IN]:clip level value. If set to -1, this feature is disabled and return to default mode.

return value :

GetColorClipLevel

declaration :
int HimageConverter::GetColorClipLevel()

parameters :

return value :

current level.

SetMonoPseudoColorMode

declaration :
void HImageConverter::SetMonoPseudoColorMode(eMonoToPseudoColorLUT mode)

description :
Force remapping of monochrome images to pseudo color, using some predefined LUT

parameters
mode[IN]: Pseudo mode. It can be set to:

`H_MONO_TO_PSEUDO_COLOR_DISABLED`: disable
`H_MONO_TO_PSEUDO_COLOR_LUT_RAINBOW`: rainbow
`H_MONO_TO_PSEUDO_COLOR_LUT_INVERTED_RAINBOW`: inverted rainbow
`H_MONO_TO_PSEUDO_COLOR_LUT_HOT`: Hot color
`H_MONO_TO_PSEUDO_COLOR_LUT_COLD`: cold color
`H_MONO_TO_PSEUDO_COLOR_LUT_NEGATIVE`: negative color

return value :

GetMonoPseudoColorMode

declaration :
eMonoToPseudoColorLUT HImageConverter::GetMonoPseudoColorMode()

description :
Get remapping monochrome images to pseudo color mode.

parameters :

return value :
Current MonoPseudoColorMode

LoadColorRemappingLUT

declaration :
void HImageConverter::LoadColorRemappingLUT(LPCTSTR fullFileStr)

description :
Load a comma separated LUT text file to define the remapping values.
The text file syntax is very simple. Each contains 4 entries: Level value to be remapped, followed with corresponding Blue, Green and Red values.

- There is no need to specify all the LUT values and entries. Only the needed value can be specified. All missing values will be interpreted as “Leave as is”,
- Level can be specified as range using [-] characters,
- # comment a line,

- Out bound values are ignored.

Example:
#Index, Blue, Green, Red Remapping value
0, 255, 0, 0 : gray level 0 will be remapping as pure blue,
1, 10, 10, ,: gray level 1 will be remapping as Blue = 10, Green = 10, Red default (1),
[200,255], 255, 255, 255: gray levels 200 to 255 are remap to pure white

parameters :

fullFileStr[in]:lut file name,if NULL disable this feature.

return value :

HIODriver

IODriver

This class allows to monitor the input line(s) of the grabber or any other supported external IO device. Call the HGrabModule::GetIODriver to access the inputs of the grabber or use the HIODriver constructor to load the *.dll* of a supported external IO device.

HIODriver

declaration :
HIODriver::HIODriver(LPCTSTR filePath)

IODriver::IODriver(string filePath)

description :
This constructor allows to load an external IO driver.

parameters :
filePath [in] The file path of the *.dll* to load. The supported *.dll* are in the Hermes\IO folder.

return value :
None.

InputDevicePresent

declaration :
bool HIODriver::InputDevicePresent()

bool IODriver::InputDevicePresent()

description :
This function checks if the device is available and connected. Make sure the device is present before calling any other functions in this class.

parameters :
None.

return value :
True if input can be used.

GetInputDeviceName

declaration :
bool HIODriver::GetInputDeviceName(LPTSTR name)

bool IODriver::GetInputDeviceName(ref string name)

description :
This function retrieves the name of the device managing the IO.

parameters :
name [in,out] The function will write the name of the device here. Allocate TCHAR [MAX_PATH].

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetInputDeviceCount

declaration :
int HIODriver::GetInputDeviceCount()

int IODriver::GetInputDeviceCount()

description :
This function retrieves the number of input devices managed by this driver. Normally, this will be 1 for grabber managed I/O Drivers, but could be more for dedicated external I/O devices.

parameters :
None.

return value :
The device count.

InitializeDeviceInput

declaration :
bool HIODriver::InitializeDeviceInput(int deviceIndex)

bool IODriver::InitializeDeviceInput(int deviceIndex)

description :
This function initializes and opens the device drivers needed to implement IO operations

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

UnInitializeDeviceInput

declaration :
bool HIODriver::UnInitializeDeviceInput(int deviceIndex)

bool IODriver::UnInitializeDeviceInput(int deviceIndex)

description :
This function uninitializes the IO device.

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetDeviceInputLineCount

declaration :
int HIODriver::GetDeviceInputLineCount(int deviceIndex)

int IODriver::GetDeviceInputLineCount(int deviceIndex)

description :
This function returns the amount of input lines available on the input device.

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

return value :
The input line count.

GetDeviceInputLineInfo

declaration :
bool HIODriver::GetDeviceInputLineInfo(int deviceIndex, int index, int* ID, LPTSTR name)

bool IODriver::GetDeviceInputLineInfo(int deviceIndex, int lineIndex, ref int ID, ref string name)

description :
Use this function to retrieve information on each available line.

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

index [in] The line index. Choose from range [1, GetInputLineCount()-1]

ID [out] The lineID used to identify the input line.

name [out] The name of the input line. Allocate TCHAR [MAX_PATH].

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetPollingDelay

declaration :
int HIODriver::GetPollingDelay(int deviceIndex, int lineID)

int IODriver::GetPollingDelay(int deviceIndex, int lineID)

description :
This function returns the current polling delay for a specific input line.

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

lineID [in] The lineID of the input line to query.

return value :
The polling delay in milliseconds.

GetCurrentLevel

declaration :
eHInputLevel HIODriver::GetCurrentLevel(int deviceIndex, int lineID)

HEnums.InputLevel IODriver::GetCurrentLevel(int deviceIndex, int lineID)

description :
This function returns the last known state of an input line. It can be either 'level-high', 'level-low' or in some cases 'unknown'.

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

lineID [in] The lineID of the input line to query.

return value :
The last known state of the input line.

Monitor

declaration :
bool HIODriver::Monitor(int deviceIndex, int lineID, InputSupportCallback callback, int

pollingDelay=100, ULONG_PTR userData=0, void* userPtr=NULL)

bool IODriver::Monitor(bool enable, int deviceIndex, int lineID, int pollingDelay, System.IntPtr userData, System.IntPtr userPtr)

description :
This function starts monitoring an input line. The callback function is called when the line state changes (rising-edge, falling-edge). To stop monitoring a line, set the callback to NULL.

parameters :
deviceIndex [in] The device index from range [0, GetInputDeviceCount()-1]

lineID [in] The lineID of the input line to monitor.

callback [in] This is a pointer to your callback function. To stop monitoring the line, set the callback to NULL. The function declaration model is as follows:

```
void HERMESCALL MyCallback(int deviceIndex, int lineID, eHInputEvent event, bool*
continueMonitoring, ULONG_PTR userData, void* userPtr);
```

The "deviceIndex" and "lineID" are the device and line on which the event is occurring (this allows to monitor multiple-lines with the same callback function). The "event" line identifies the event detected (rising-edge or falling-edge). "continueMonitoring" can be set to false if the user does not want to be notified about events on this line. "userData" and "userPtr" are the same as the values passed to the Monitor function.

pollingDelay [in] This value determines the frequency, in milliseconds, at which the line will be queried. Using a low value will require more resources while using a high value could omit edges.

userData [in] This value will be sent back in the callback function.

userPtr [in] This pointer will be sent back in the callback function.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

PulseGeneratorPresent

declaration :
bool HIODriver::PulseGeneratorPresent()

description :
This function checks if the pulse generator is available for the current grabber. Make sure the pulse generator is present before calling any other functions related to it.

parameters :
None.

return value :

True if pulse generator can be used.

InitializePulseGenerator

declaration :
bool HIODriver::InitializePulseGenerator()

description :
This function initializes and opens the pulse generator.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

UnInitializePulseGenerator

declaration :
bool HIODriver::UnInitializePulseGenerator()

description :
This function uninitializes the pulse generator.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StartPulseGenerator

declaration :
bool HIODriver::StartPulseGenerator(double freq)

description :
This function starts the pulse generator with a specific frequency.

parameters :
freq [in] The frequency for the pulse generator.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StopPulseGenerator

declaration :
bool HIODriver::StopPulseGenerator()

description :
This function stops the pulse generator.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

.NET Callback Mechanism (.NET only)

The following delegates are used to create the callback functions.

delegate void OnIODriverInputDelegate(int deviceIndex, int lineID, HEnums.InputEvent ioEvent, ref bool continueMonitoring, System.IntPtr userData, System.IntPtr userPtr)

SetCallback

declaration :
void IODriver::SetCallback(OnIODriverInputDelegate callback)

description :
This function set the callback function called when an event occurs on a monitored line.

parameters :
func [in] The callback function.

return value :
None.

HFPSMonitor

This module can be used to easily compute a frame rate. It can be used to monitor the frames per second speed of anything, be it capture, playback or even non image related content.

Typical usage :

```
HFPSMonitor FPSMonitor;

//Set the function to be called at each fps update.
FPSMonitor.SetCallbacks(...);

//Set the FPS monitor to update us with the new fps result every 1000 ms
FPSMonitor.StartLiveCheck(1000);

//Every time a new frame is received from the grabber, tell the monitor
FPSMonitor.NewLiveFrame(false); //new frame received
```

StartLiveCheck

declaration :
bool HFPSMonitor::StartLiveCheck(unsigned int updateInterval = 1000)

bool FPSMonitor::StartLiveCheck(unsigned int updateInterval)

description :
Updates the FPS at the desired milliseconds interval. Every *updateInterval*, the module will launch the *OnNewFrame* callback. (see the *SetCallbacks* function)

parameters :
updateInterval [in] The update interval in milliseconds.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

NewLiveFrame

declaration :
bool HFPSMonitor::NewLiveFrame(bool forceUpdate)

bool FPSMonitor::NewLiveFrame(bool forceUpdate)

description
Transmits to the monitor that a new frame was just received. The monitor will then recompute the FPS value and launch the *OnNewFrame* callback if the *updateInterval* time is reached.

parameters :
forceUpdate [in] If true, the *OnNewFrame* callback will be launch regardless of the *updateInterval* set.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetCallbacks

declaration :
bool HFPSMonitor::SetCallbacks(HFPSMonitorCallbacks* userCallbacks, unsigned int userData=0, void* userPtr=NULL)

description :
SetCallbacks sets the callback class used by the grabbing module. (see the *HFPSMonitorCallbacks* class)

parameters :
userCallbacks [in] The callbacks class to use. This will be a pointer to one of the user's *HFPSMonitorCallbacks*-derived class.

userData [in] Use this parameter to pass a value to be used in your callback functions.

userPtr [in] Use this parameter to pass a pointer to be used in your callback functions.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StartBench

declaration :
bool HFPSMonitor::StartBench()

bool FPSMonitor::StartBench()

description :
Used in conjunction with *EndBench*, *StartBench* is used to compute the frame rate over a given time period. Call *StartBench* to save the current time.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

EndBench

declaration :
bool HFPSMonitor::EndBench(unsigned long frameCount, double& fps)

bool FPSMonitor::EndBench(unsigned long frameCount, double% fps)

description :
StartBench must have been called at least once prior to calling *EndBench*. This function will compute the frame rate given a specific number of frames.

parameters :
frameCount [in] The number of frames captured since *StartBench* has been called.

fps [out] The frame rate. fps = *frameCount* / ((*EndBench* Time) - (*StartBench* Time)).

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetBenchTime

declaration :
bool HFPSMonitor::GetBenchTime(long& seconds, unsigned short& ms)

bool FPSMonitor::GetBenchTime(long% seconds, unsigned short% ms)

description :
StartBench must have been called at least once prior to calling *GetBenchTime*. This function will compute the time elapsed since the bench was started.

parameters :
seconds [out] The number of seconds elapsed since *StartBench* has been called.

ms [out] The milliseconds part of the time elapsed.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

.NET Callback Mechanism (.NET only)

The following delegate is used to create the callback function.

public delegate void OnFramerateNotificationDelegate(double framerate, int userData, IntPtr userPtr);

SetCallbackData

declaration :
void FPSMonitor::SetCallbackData(unsigned int data)

description :
This function set the user data member than will be returned as a parameter of the callback functions.

parameters :
data [in] An unsigned integer.

return value :
None.

SetCallbackPtr

declaration :
void FPSMonitor::SetCallbackPtr(IntPtr ptr)

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
ptr [in] A pointer.

return value :
None.

SetCallbackOnFramerate

declaration :
void FPSMonitor::SetCallbackOnFramerate(OnFramerateNotificationDelegate^ func)

description :
This function set the callback function called every time a new fps value is ready.

parameters :

func [in] the function delegate.

return value :
None.

HFPSMonitorCallbacks

This class allows the user to be informed when certain events occurs in the *HFPSMonitor* module, by simply making one of the user's class inherit from *HFPSMonitorCallbacks* and overriding the functions needing to be custom processed.

For example :

```
CMyClass : public HFPSMonitorCallbacks
{
private:
        //custom code...

public:
        virtual void OnNewFrame(double fps, unsigned int userData,
  void* userPtr)
        {
                //The FPS monitor is sending us a “frame per second” update.
                //We could update the display of the grabber speed shown
                //in the user interface.
                .......
        };

        //more custom code...
};
```

Then use the *HFPSMonitor*'s *SetCallbacks* function to set this class to be called instead of the inactive default function.

OnNewFrameRate

declaration :
virtual void HFPSMonitorCallbacks::OnNewFrameRate(double fps, unsigned int userData, void* userPtr)

description :
This function is called by the FPS monitor module when the *updateInterval* is reached or if instructed to do so by a call to *HFPSMonitor*::*NewLiveFrame(true).*

parameters :
fps [out] The new speed in frames per second.

userData [out] The value set in the SetCallback function.

userPtr [out] The value set in the SetCallback function.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call *GetLastError* to retrieve the error code.

HImage

HermesImage

The *HImage* is an image container. It comprises the data of a single image and offers several functions to query the image format. Some modules create *HImage* as their outputs, like *HGrabModule* (images captured from the camera) or *HSequencer* (images read from a sequence during browsing or playback). Other modules, like *Hviewer,* use *HImages* as inputs.

CreateHImage

declaration :
static HImage* HImage::CreateHImage()

static HermesImage HermesImage::CreateHermesImage()

description :
CreateHimage creates an empty Himage in which one image from a sequence file can be read. When it is no longer needed, free it by using *DeleteHimage*.

parameters :
None.

return value :
A pointer to a newly created HImage.

example :

see HSequencer::Read()

CloneHImage

declaration :
static HImage* HImage::CloneHImage(HImage* srcImage)

static HermesImage CloneHermesImage(HermesImage srcImage)

description :
This function creates a perfect copy of a source *HImage*. When it is no longer needed, free it by using *DeleteHimage*.

parameters :
srcImage [in] The source image to clone.

return value :
A pointer to the newly created *HImage*.

CloneHImage

declaration :
static void HImage::CloneHImage(HImage* srcImage, HImage* destImage)

static void HermesImage::CloneHermesImage(HermesImage srcImage, HermesImage destImage)

description :
This function creates a perfect copy of the srcImage in the destImage. For the destImage, only use HImage you created using HImage::CreateImage.

parameters :
srcImage [in] The source image to clone.

destImage [in] The destination image.

return value :
None.

DeleteHImage

declaration :
static void HImage::DeleteHImage(HImage* image)

static void HermesImage::DeleteHermesImage(HermesImage image)

description :
DeleteHImage deletes an *HImage* allocated within a *CreateHImage*. Only user created *Himage*s should be deleted

parameters :
image [in] The *HImage* to delete.

return value :
None.

example :
see HSequencer::Read()

ReallocateImage

declaration :
void HImage::ReallocateImage(unsigned int width, unsigned int height, unsigned int bitDepth, unsigned int bitDepthReal, eHImageFormat imageFormat)

void HermesImage::ReallocateImage(uint width, uint height, uint bitDepth, uint bitDepthReal, HEnums.ImageFormat imageFormat)

description :
ReallocateImage will delete the image currently stored in the *HImage* and will reallocate memory to fit the new image following the specified parameters. Only user created *Himage*s should be reallocated.

parameters :
width [in] The width of the *HImage* (in pixel).

height [in] The height of the *HImage* (in pixel).

bitDepth [in] The bit depth of the *HImage*.

bitDepthReal [in] The real bit depth of the *HImage*. See the description in the *Definitions* section near the beginning of this document.

imageFormat [in] The image format. Available formats are shown in *HEnums.h*. As formats ending with _JPEG are sequence formats, not image formats, they should not be used here.

return value :
None.

GetImageWidth

declaration :
unsigned int HImage::GetImageWidth()

uint HermesImage::GetImageWidth()

description :
GetImageWidth retrieves the width in pixels of the image.

parameters :
None.

return value :
The width of the image in pixels.

GetImageHeight

declaration :
unsigned int HImage::GetImageHeight()

uint HermesImage::GetImageHeight()

description :
GetImageHeight retrieves the height in pixels of the image.

parameters :
None.

return value :
The height of the image in pixels.

GetImageBitDepth

declaration :
unsigned int HImage::GetImageBitDepth()

uint HermesImage::GetImageBitDepth()

description :
GetImageBitDepth retrieves the bit depth of the image.

parameters :
None.

return value :
The function will return the bit depth of the image. If the *HImage* is empty, the returned value is 0.

GetImageBitDepthReal

declaration :
unsigned int HImage::GetImageBitDepthReal()

uint HermesImage::GetImageBitDepthReal()

description :
GetImageBitDepthReal retrieves the real bit depth of the image.

parameters :
None.

return value :
The function will return the real bit depth of the images. If the *HImage* is empty, the returned value is 0.

GetImageSizeBytes

declaration :
unsigned int HImage::GetImageSizeBytes()

uint HermesImage::GetImageSizeBytes()

description :

Retrieves the size of the image in bytes.

parameters :
None.

return value :
The size of the image in bytes. If the *HImage* is empty, the returned value is 0.

GetImageFormat

declaration :
eHImageFormat HImage::GetImageFormat()

HEnums.ImageFormat HermesImage::GetImageFormat()

description :
GetImageFormat retrieves the format of the image.

parameters :
None.

return value :
The function will return the format of the captured images. If no grabber is selected or if the format can not be determined, the returned value is *H_IMAGE_UNKNOWN.*

GetRawImageData

declaration :
void* HImage::GetRawImageData()

System.IntPtr HermesImage::GetRawImageData()

description :
Retrieves a pointer to the actual image data, as grabbed from the grabber or read from a sequence. For monochrome or color packed images, this points to the full image data. In the case of a planar color image (format is H_IMAGE_PLANAR), use the GetRedChannel, GetGreenChannel and GetBlueChannel functions to access the individual color channels.

parameters :
None.

return value :
A pointer to the image data or red channel for planar images. If the *HImage* is empty, the returned value is NULL.

GetRedChannel

declaration :
void* HImage::GetRedChannel()

System.IntPtr HermesImage::GetRedChannel()

description :
Retrieves a pointer to the red image data of a planar image. You can also use GetRawImageData() to access the whole image data (red/green/blue).

parameters :
None.

return value :
A pointer to the image data or red channel for planar images. If the *HImage* is empty, the returned value is NULL.

GetGreenChannel

declaration :
void* HImage::GetGreenChannel()

System.IntPtr HermesImage::GetGreenChannel()

description :
Retrieves a pointer to the green image data of a planar image. If the image format is not planar, the green channel is empty.

parameters :
None.

return value :
A pointer to the green channel. If the *HImage* is empty or if the image is not planar, the returned value is NULL.

GetBlueChannel

declaration :
void* HImage::GetBlueChannel()

System.IntPtr HermesImage::GetBlueChannel()

description :
Retrieves a pointer to the actual image data (as grabbed from the grabber or read from a sequence). In the case of a planar image, it will return a pointer to the red channel data. For monochrome or color packed images, this points to the full image data.

parameters :
None.

return value :
A pointer to the blue channel. If the *HImage* is empty or if the image is not planar, the returned value is NULL.

GetTimestamp

declaration :
long HImage::GetTimestamp()

int HermesImage::GetTimestamp()

description :
GetTimestamp retrieves the timestamp of the image. The timestamp indicates the exact time at which the image was captured. More precision can be obtained by calling *GetTimestampMS* to get the milliseconds part.

parameters :
None.

return value :
The absolute timestamp in seconds.

example :

```
//Print the timestamp of the HImage "image";
long t = image->GetTimestamp();
printf("timestamp : %.19s.%.003hu", _tctime(&t), image->GetTimestampMS());
```

GetTimestampMS

declaration :
unsigned short HImage::GetTimestampMS()

ushort HermesImage::GetTimestampMS()

description :
This will retrieve the milliseconds part of the absolute image timestamp. Depending on the latency of the operating system, the precision might be off by 1 to 3 milliseconds.

parameters :
None.

return value :
The milliseconds part of the image timestamp.

example :

see HImage::GetTimestamp()

GetTimestampUS

declaration :
unsigned shot HImage::GetTimestampUS()

ushort HermesImage::GetTimestampUS()

description :
The microseconds part of the absolute image timestamp. This value will be 0 unless the frame was timestamped at while using an external high performance timer device in conjunction with the grabber..

parameters :
None.

return value :
The microsecond part of the image timestamp.

SetTimestamp

declaration :
void HImage::SetTimestamp(long timeStamp, unsigned short timeStampMS, unsigned short timeStampUS)

void HermesImage::SetTimestamp(int timeStamp, ushort timeStampMS, ushort timeStampUS)

description :
This function change the timestamp of a HImage.

parameters :
timeStamp [in] The absolute timestamp in seconds.

timeStampMS [in] The milliseconds part of the image timestamp.

timeStampUS [in] The microsecond part of the image timestamp.

return value :
None.

GetMetadataCount

declaration :
int HImage::GetMetadataCount()

description :
Retrieve the number of metadata attached to this image.

parameters :
None.

return value :
The metadata count.

GetMetadata

declaration :
HMetadata* HImage::GetMetadata(int index)

description :
Get one of the metadata attached to the image.

parameters :
index [in] The index of the metadata,valid range is : [0, GetMetadataCount()-1].

return value :
The metadata.

GetSpecificMetadataCount

declaration :
int HImage::GetSpecificMetadataCount(unsigned int uID)

description :
Retrieve the number of metadata of a specific type attached to this image.

parameters :
uID [in] The unique identifier of the metadata type to retrieve.

return value :
The metadata count.

GetSpecificMetadata

declaration :
HMetadata* HImage::GetSpecificMetadata(unsigned int uID, int index)

description :
Get one of the metadata of a specific type attached to the image.

parameters :
uID [in] The unique identifier of the metadata type to retrieve.
index [in] The index of the metadata,valid range is : [0, GetSpecificMetadataCount()-1].

return value :
The metadata.

AddMetadata

declaration :
void HImage::AddMetadata(const HMetadata& metadata)

description :
Add metadata to the image.

parameters :
metadata [in] The metadata to add.

return value :
None.

RemoveAllMetadata

declaration :
void HImage::RemoveAllMetadata()

description :
Remove all the metadatas attached to the image, if any.

parameters :
None.

return value :
None.

HasImageDecoded

declaration :
`void` HImage::HasImageDecoded()

description :
Get if image has been decoded if image is H264 frame. Other compression type is ignored always True.

parameters :
None.

return value :
True if the image has decoded else False.

HermesUtils

HermesUtils

The Hermes *Utils* is a set of utility functions.

HImageFormatString

declaration :
LPCTSTR HImageFormatString(eHImageFormat format)

static string HermesUtils::ImageFormatToString(HEnums.ImageFormat format)

description :
Use this function to get a string describing a particular image format. For example, format H_IMAGE_MONO will be translated to "Monochrome image".

parameters :
format [in] The image format.

return value :
A string with the translated name.

AbsoluteToRelative

declaration :
void AbsoluteToRelative(long reference, unsigned short referenceMS, unsigned short referenceUS, long absolute, unsigned short absoluteMS, unsigned short absoluteUS, unsigned long& relative, unsigned short& relativeMS, unsigned short& relativeUS)

static void HermesUtils::AbsoluteToRelativeTime(int reference, ushort referenceMS, ushort referenceUS, int absolute, ushort absoluteMS, ushort absoluteUS, ref uint relative, ref ushort relativeMS, ref ushort relativeUS)

description :
This function converts an absolute time into a relative time. Normally, the reference frame will be the first frame of the sequence.

parameters :
reference [in] Absolute time of reference frame.

referenceMS [in] Absolute time of reference frame. Millisecond part.

referenceUS [in] Absolute time of reference frame. Microsecond part.

absolute [in] Absolute time to convert.

absoluteMS [in] Absolute time to convert. Millisecond part.

absoluteUS [in] Absolute time to convert. Microsecond part.

relative [out] Converted relative time.

relativeMS [out] Converted relative time. Millisecond part.

relativeUS [out] Converted relative time. Microsecond part.

return value :
None.

RelativeToAbsolute

declaration :
void RelativeToAbsolute(long reference, unsigned short referenceMS, unsigned short referenceUS, unsigned long relative, unsigned short relativeMS, unsigned short relativeUS, long& absolute, unsigned short& absoluteMS, unsigned short& absoluteUS)

static void HermesUtils::RelativeToAbsoluteTime(int reference, ushort referenceMS, ushort referenceUS, uint relative, ushort relativeMS, ushort relativeUS, ref int absolute, ref ushort absoluteMS, ref ushort absoluteUS)

description :
This function converts a relative time to an absolute time. Most of the time, the reference frame will be the first frame of the sequence.

parameters :
reference [in] Absolute time of reference frame.

referenceMS [in] Absolute time of reference frame. Millisecond part.

referenceUS [in] Absolute time of reference frame. Microsecond part.

relative [in] Relative time to convert.

relativeMS [in] Relative time to convert. Millisecond part.

relativeUS [in] Relative time to convert. Microsecond part.

absolute [out] Converted absolute time.

absoluteMS [out] Converted absolute time. Millisecond part.

absoluteUS [out] Converted absolute time. Microsecond part.

return value :
None.

ErrorCodeToString

declaration :
static string HermesUtils::ErrorCodeToString(HEnums.ErrorCode code)

description :
This function return a string explaining an error code.

parameters :
code [in] The error code returned by a GetLastError() function.

return value :
A string with the error explanation.

HImageColorProcessing

This module can be used to apply a Bayer conversion on an raw image and/or to color balance an image. This class replaces HbayerConverter now obsolete.

ApplyColorProcessing

declaration :
bool HImageColorProcessing::ApplyColorProcessing(HImage* in, HImage* out, bool* wasDoneInPlace)

bool ImageColorProcessing::ApplyColorProcessing(HermesImage^ in, HermesImage^ out, bool% wasDoneInPlace)

description :
This function apply the selected color processing on an image. The conversion applied are selected by calling *EnableBayerConversion* and *EnableColorBalance*.

parameters :
in [in, out] The source image to be processed (must be a valid HImage).

out [in, out] The processed image. (must be a valid HImage, possibly created with HImage::CreateHImage())

wasDoneInPlace [out] True If the color processing could be done directly on the source image (in that case the *out* is not used). False if *out* had to be used (ex : Bayer conversion cannot be done in-place).

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsBayerConversionEnabled

declaration :
bool HImageColorProcessing::IsBayerConversionEnabled()

bool ImageColorProcessing::IsBayerConversionEnabled()

description :
Check if the module will apply Bayer conversion on source images.

parameters :
None.

return value :
True if Bayer conversion is to be applied.

EnableBayerConversion

declaration :
void HImageColorProcessing::EnableBayerConversion(bool enable)

void ImageColorProcessing::EnableBayerConversion(bool enable)

description :
Enable or disable the Bayer conversion.

parameters :
enable [in] True to enable the the Bayer conversion. False to disable it.

return value :
None.

GetPatternOrigin

declaration :
eHBayerPatternOrigin HImageColorProcessing::GetPatternOrigin()

HEnums::BayerPatternOrigin ImageColorProcessing::GetPatternOrigin()

description :
Get the current Bayer pattern used for the conversion.

parameters :
None.

return value :
The current Bayer pattern.

SetPatternOrigin

declaration :
void HImageColorProcessing::SetPatternOrigin(eHBayerPatternOrigin value)

void ImageColorProcessing::SetPatternOrigin(HEnums::BayerPatternOrigin value)

description :
Instruct the module about the Bayer pattern used in the source image.

parameters :
value [in] The pattern to use for conversion.

return value :
None.

IsBGRxConversionEnabled

declaration :
bool HImageColorProcessing::IsBGRxConversionEnabled()

bool ImageColorProcessing::IsBGRxConversionEnabled()

description :
Check the Bayer conversion output format.

parameters :
None.

return value :
True if the Bayer conversion output a BGRx format instead of BGR.

EnableBGRxConversion

declaration :
void HImageColorProcessing::EnableBGRxConversion(bool enable)

void ImageColorProcessing::EnableBGRxConversion(bool enable)

description :
The Bayer conversion output by default a BGR image. However, it is possible to force a BGRx format.

parameters :
enable [in] True for BGRx, false for BGR.

return value :
None.

IsMonochromeConversionEnabled

declaration :
bool HImageColorProcessing::IsMonochromeConversionEnabled()

bool ImageColorProcessing::IsMonochromeConversionEnabled()

description :
Get output image mode as monochrome or color.

parameters :
None.

return value :
True if the Bayer conversion output a BGRx format instead of BGR.

EnableMonochromeConversion

declaration :
void HImageColorProcessing::EnableMonochromeConversion(bool enable)

void ImageColorProcessing::EnableMonochromeConversion(bool enable)

description :
If source image is monochrome and Bayer conversion disabled, this function force output image format as monochrome

parameters :
enable [in] True for force to monochrome

return value :
None.

IsColorProcessingHQConversionEnabled

declaration :
bool HImageColorProcessing::IsColorProcessingHQConversionEnabled()

bool ImageColorProcessing::IsColorProcessingHQConversionEnabled()

description :
Get high bit depth Bayer conversion precision mode.

parameters :
None.

return value :
True if the Bayer conversion convert >8 bits raw images to 48 bits BGR.

EnableColorProcessingHQConversion

declaration :
void HImageColorProcessing::EnableColorProcessingHQConversion(bool enable)

void ImageColorProcessing::EnableColorProcessingHQConversion(bool enable)

description :
When the source image bit depth is greater than 8bpp it is possible to force the Bayer conversion output to be in BGR48 instead of default BGR24. This setting has no effect if the Bayer conversion

outputs BGRx (see EnableBayerConversionToBGRx()).

parameters :
enable [in] True to convert to BGR48 as needed.

return value :
None.

GetSampleFactor

declaration :
eHBayerSampleFactor HImageColorProcessing::GetSampleFactor()

HEnums::BayerSampleFactor ImageColorProcessing::GetSampleFactor()

description :
Get the current sample factor for the Bayer conversion.

parameters :
None.

return value :
The current sample factor.

SetSampleFactor

declaration :
void HImageColorProcessing::SetSampleFactor(eHBayerSampleFactor value)

void ImageColorProcessing::SetSampleFactor(HEnums::BayerSampleFactor value)

description :
Set the sample factor applied during the Bayer conversion. Default is 1:1. However, 1:2 (half the pixels), 1:4 (quarter of the pixels) and 1:8 are also available. This allows to apply the Bayer conversion faster. This is useful to quickly render an image for preview. The output images will be smaller than the source image. The sample factor is applied only if the calculation mode is H_BAYER_CALCMODE_FAST.

parameters :
value [in] The sample factor value.

return value :
None.

GetCalcMode

declaration :

eHBayerCalculationMode HImageColorProcessing::GetCalcMode()

HEnums::BayerCalculationMode ImageColorProcessing::GetCalcMode()

description :
Get the current Bayer conversion calculation mode.

parameters :
None.

return value :
The current calc mode.

SetCalcMode

declaration :
void HImageColorProcessing::SetCalcMode(eHBayerCalculationMode value)

void ImageColorProcessing::SetCalcMode(HEnums::BayerCalculationMode value)

description :
The calculation mode affect the accuracy of the Bayer conversion. More precise calculations require more CPU time.

parameters :
value [in] The calculation mode to use.

return value :
None.

IsMatrixColorCorrectionEnabled

declaration :
bool HImageColorProcessing::IsMatrixColorCorrectionEnabled()

bool ImageColorProcessing::IsMatrixColorCorrectionEnabled()

description :
Get Matrix correction mode (default: off)

parameters :
None.

return value :
True if the color Matrix correction is enabled.

EnableMatrixColorCorrection

declaration :
void HImageColorProcessing::EnableMatrixColorCorrection(bool enable)

void ImageColorProcessing::EnableMatrixColorCorrection(bool enable)

description :
Enable the color Matrix correction mode.

parameters :
enable [in] True to enable the color Matrix correction.

return value :
None.

GetCorrectionMatrix

declaration :
void HImageColorProcessing::GetCorrectionMatrix(double& factorBlueFromBlue, double& factorBlueFromGreen, double& factorBlueFromRed, double& factorBlaue,double& factorGreenFromBlue, double& factorGreenFromGreen, double& factorGreen,double& factorGreenFromRed, double& factorRedFromBlue, double& factorRedFromGreen, double& factorRedFromRed,double& factorRed)

void ImageColorProcessing::GetCorrectionMatrix(double% factorBlueFromBlue, double% factorBlueFromGreen, double% factorBlueFromRed, double% factorBlue,double% factorGreenFromBlue, double% factorGreenFromGreen, double% factorGreenFromRed,double% factorGreen, double% factorRedFromBlue, double% factorRedFromGreen, double% factorRedFromRed,double% factorRed)

description :
Get the values of the color matrix that is applied if the Bayer color correction is enabled.

parameters :
factorXFromX [out] The various values in the correction matrix.

return value :
None.

SetCorrectionMatrix

declaration :
void HImageColorProcessing::SetCorrectionMatrix(double factorBlueFromBlue, double factorBlueFromGreen, double factorBlueFromRed, double factorBlue,double factorGreenFromBlue, double factorGreenFromGreen, double factorGreenFromRed, double factorgreen,double factorRedFromBlue, double factorRedFromGreen, double factorRedFromRed,double factorRed)

void ImageColorProcessing::SetCorrectionMatrix(double factorBlueFromBlue, double factorBlueFromGreen, double factorBlueFromRed, double factorBlue,double factorGreenFromBlue, double factorGreenFromGreen, double factorGreenFromRed, double factorGreen,double factorRedFromBlue, double factorRedFromGreen, double factorRedFromRed,double factorRed)

description :
Set the values for the color matrix that is applied if the Bayer color correction is enabled.

parameters :
factorXFromX [in] The various values required by the correction matrix.

return value :
None.

IsLUTProcessEnabled

declaration :
bool HImageColorProcessing::IsLUTProcessEnabled()

bool ImageColorProcessing::IsLUTProcessEnabled()

description :
Check if a LUT processing is applied on processed images.

parameters :
None.

return value :
True if the color balance is enabled.

EnableLUTProcess

declaration :
void HImageColorProcessing::EnableLUTProcess(bool enable)

void ImageColorProcessing::EnableLUTProcess(bool enable)

description :
Enable or disable the application of a LUT processing on processed images. If enabled along with the Bayer conversion, the LUT processing will be applied after (or at the same time as) the Bayer conversion.

parameters :
enable [in] *True* to apply the LUT processing processing.

return value :

None.

IsAutoColorBalanceEnabled

declaration :
bool HImageColorProcessing::IsAutoColorBalanceEnabled()

bool ImageColorProcessing::IsAutoColorBalanceEnabled()

description :
Check if the color balance is automatic or manual.

parameters :
None.

return value :
True if the color balance is automatic. *False* if manual.

EnableAutoColorBalance

declaration :
void HImageColorProcessing::EnableAutoColorBalance(bool enable)

void ImageColorProcessing::EnableAutoColorBalance(bool enable)

description :
Set the type of color balance applied. Automatic or manual are available. The default is manual.

parameters :
enable [in] *True* to apply automatic color balance, *false* for manual color balance.

return value :
None.

GetAutoColorBalanceAlgorithm

declaration :
eHAutoWhiteBalanceAlgorithm HImageColorProcessing::GetAutoColorBalanceAlgorithm()

HEnums::AutoWhiteBalanceAlgorithm ImageColorProcessing::GetAutoColorBalanceAlgorithm()

description :
Get the automatic color balance algorithm applied.

parameters :
None.

return value :
The color balance algorithm.

SetAutoColorBalanceAlgorithm

declaration :
void HImageColorProcessing::SetAutoColorBalanceAlgorithm(eHColorBalanceAlgorithm value)

void ImageColorProcessing::SetAutoColorBalanceAlgorithm(HEnums:: ColorBalanceAlgorithm value)

description :
Set the automatic color balance algorithm.

parameters :
value [in] The algorithm to use.

return value :
None.

GetManualColorBalance

declaration :
void HImageColorProcessing::GetManualColorBalance(int& red, int& green, int& blue)

void ImageColorProcessing::GetManualColorBalance(int% red, int% green, int% blue)

description :
Get the color balance offset values applied in manual mode. Values are in percent of the full image dynamic range.

parameters :
red [out] Offset value applied to the red component.

green [out] Offset value applied to the green component.

blue [out] Offset value applied to the blue component.

return value :
None.

SetManualColorBalance

declaration :
void HImageColorProcessing::SetManualColorBalance(int red, int green, int blue)

void ImageColorProcessing::SetManualColorBalance(int red, int green, int blue)

description :
Set the color balance offset values applied when in manual mode. Values are in percent of the full image dynamic range.

parameters :
red [in] Offset value applied to the red component.

green [in] Offset value applied to the green component.

blue [in] Offset value applied to the blue component.

return value :
None.

RecalculateLUT

declaration :
void HImageColorProcessing::RecalculateLUT()

void ImageColorProcessing::RecalculateLUT()

description :
Force the reinitialization of the LUT for LUT Processing.

parameters :
None.

return value :
None.

Is3DLookupTable

declaration :
void HImageColorProcessing::Is3DLookupTable()

description :
Check if 3D LUT mode is enabled

parameters :
None.

return value :
True if 3D LUT enabled else 1D LUT enabled.

Enable3DLookupTable

declaration :
void HimageColorProcessing::Enable3DLookupTable(bool d3dlut)

description :
Choose 3D LUT for 1D LUT

parameters :
d3dlut[in]true choose 3D LUT else 1D LUT.

return value :

SetAutoWhiteBalanceROI

declaration :
void SetAutoWhiteBalanceROI(double offsetX,double offsetY,double width,double height);

description :
Set an ROI to calculate automatic White Balance parameters. Default is full image

parameters :
offsetX [in] The horizontal offset percentage of the region.
offsetY [in] The vertical offset percentage of the region.
sizeX [in] The width percentage of the region.
sizeY [in] The height percentage of the region.

return value :

GetAutoWhiteBalanceROI

declaration :
void GetAutoWhiteBalanceROI(double& offsetX,double& offsetY,double& width,double& height);

description :
Return ROI of calculation for auto WhiteBalance.

parameters :
offsetX [out] The horizontal offset percentage of the region.
offsetY [out] The vertical offset percentage of the region.
sizeX [out] The width percentage of the region.
sizeY [out] The height percentage of the region.

ResetASCSettings

declaration :
void HImageColorProcessing::`ResetASCSettings`()

description :
Reset ASC CDL settings to default value.
American Society of Cinematographers Color Decision List (ASC CDL). This define a format for the exchange of basic primary color grading information. A 3 parameters formula can be used to define most possible remapping and color correction:

<u>The formula for ASC CDL color correction is:</u>

where:

- *out* is the color graded pixel code value
- *i* is the input pixel code value (0=black, 1=white)
- *s* is slope (any number 0 or greater, nominal value is 1.0)
- *o* is offset (any number, nominal value is 0)
- *p* is power (any number greater than 0, nominal value is 1.0)

The formula is applied to the three color values for each pixel using the corresponding slope, offset, and power(gamma) for each color channel.

Parameters

return value :

GetASCSettingsParam

declaration :
bool HImageConverter::GetASCSettingsParam(double ascSlope[3],double ascOffset[3], double ascPower[3])

description :
Get current ASC parameters settings. Array Index 0 is Blue channel ;1 is Green channel; 2 is Red channel. When processing monochrome image use index 0 only.

parameters :

ascSlope [out]: ASC slope parameter.
ascOffset [out]: ASC offset parameter.
ascPower [out]: ASC power parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetASCSettingsParam

declaration :
bool HImageColorProcessing::SetASCSettingsParam(double ascSlope[3],double ascOffset[3], double ascPower[3])

description :
Set ASC Parameter. Array Index 0 is Blue channel ;1 is Green channel; 2 is red channel. When processing monochrome image use index 0 only.

parameters :
ascSlope [IN]: ASC slope parameter.
ascOffset [IN]: ASC offset parameter.
ascPower [IN]: ASC power parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetColorClipLevel

declaration :
void HImageColorProcessing::SetColorClipLevel(int level)

description :
This function is only available for pixel depth greater than 8 bits. By default, HImageConvert always return the 8 most significant bits of captured images. This function can highlight specific bit ranges on the images.

As an example, a 10 bit image can be leveled in 3 ways:

- level 2: 8 most significant bits (bit 2 to 10, default HImageConvert setting)
- level 1: 8 middle significant bits (bit 1 to 9)
- level 0: 8 least significant bits (bit 0 to 7)

parameters :
level[IN]:clip level value. If set to -1, this feature is disabled and return to default mode.

return value :

GetColorClipLevel

declaration :
int HImageColorProcessing::GetColorClipLevel()

parameters :

return value :
current level.

SetMonoPseudoColorMode

declaration :
void HImageColorProcessing::SetMonoPseudoColorMode(eMonoToPseudoColorLUT mode)

description :
Force remapping of monochrome images to pseudo color, using some predefined LUT.

parameters
mode[IN]: Pseudo mode. It can be set to:

- `H_MONO_TO_PSEUDO_COLOR_DISABLED`: disable
- `H_MONO_TO_PSEUDO_COLOR_LUT_RAINBOW`: rainbow
- `H_MONO_TO_PSEUDO_COLOR_LUT_INVERTED_RAINBOW`: inverted rainbow
- `H_MONO_TO_PSEUDO_COLOR_LUT_HOT`: Hot color
- `H_MONO_TO_PSEUDO_COLOR_LUT_COLD`: cold color
- `H_MONO_TO_PSEUDO_COLOR_LUT_NEGATIVE`: negative color

return value :

GetMonoPseudoColorMode

declaration :
eMonoToPseudoColorLUT HImageColorProcessing::GetMonoPseudoColorMode()

description :
Get remapping monochrome images to pseudo color mode.

parameters :

return value :
Current MonoPseudoColorMode

ExportLUTToTextFile

declaration :
bool HImageColorProcessing::ExportLUTToTextFile(LPCTSTR fullFileStr)

description :
Load a comma separated LUT text file to define the remapping values.
The text file syntax is very simple. Each contains 4 entries: Level value to be remapped, followed with corresponding Blue, Green and Red values.

- There is no need to specify all the LUT values and entries. Only the needed value can be specified. All missing values will be interpreted as “Leave as is”,
- Level can be specified as range using [-] characters,
- # comment a line,
- Out bound values are ignored.

Example:
#Index, Blue, Green, Red Remapping value
0, 255, 0, 0 : gray level 0 will be remapping as pure blue,
1, 10, 10, ,: gray level 1 will be remapping as Blue = 10, Green = 10, Red default (1),
[200,255], 255, 255, 255: gray levels 200 to 255 are remap to pure white

parameters :
fullFileStr[in]:lut file name,if NULL disable this feature.

return value :
True if the function is successful or *false* if the function failed.

LoadNpxTextLutFile

declaration :
void HImageColorProcessing::LoadNpxTextLutFile(LPCTSTR fullFileStr)

description :
See ExportLUTToTextFile

parameters :
fullFileStr[in]:lut file name,if NULL disable this feature.

return value :
True if the function is successful or *false* if the function failed.

SetCacheLUTFileName

declaration :
void HImageColorProcessing::SetCacheLUTFileName(LPCTSTR fullFileStr,bool reload)

description :
Set destination path and file name to cache current LUT data. LUT processing store current LUT data to a file for future use to avoid unneeded recalculation.

parameters :
fullFileStr[in]: cache lut file name. if NULL disable this feature.
reload[in]:True immediately reload and use LUT data contained in the fullFileStr file.

return value :

Get1DSupportThirdPartLutFileCount

declaration :
static void HImageColorProcessing::Get1DSupportThirdPartLutFileCount()

description :
Get current support 1D third part LUT file format count.

parameters :

return value :
numbers of support 1D third part LUT file format

Get1DSupportThirdPartLutFileFormatInfo

declaration :
static void HimageColorProcessing::Get1DSupportThirdPartLutFileFormatInfo(int index, LPTSTR desc,LPTSTR ext);

description :
Get support 1D third part LUT file format infomation.

parameters :
index [in] the index of file format to query
desc[out] description for for LUT file format
ext[out] extension name of the file

return value :
True if the function is successful or *false* if the function failed.

Get3DSupportThirdPartLutFileCount

declaration :
static void HImageColorProcessing::Get3DSupportThirdPartLutFileCount()

description :
Get current support 3D third part LUT file format count.

parameters :

return value :
numbers of support 3D third part LUT file format

Get3DSupportThirdPartLutFileFormatInfo

declaration :
static void HimageColorProcessing::Get3DSupportThirdPartLutFileFormatInfo(int index, LPTSTR desc,LPTSTR ext);

description :
Get support 3D third part LUT file format infomation.

parameters :
index [in] the index of file format to query
desc[out] description for for LUT file format
ext[out] extension name of the file

return value :
True if the function is successful or *false* if the function failed.

AddThirdPartLutFile

declaration :
bool HimageColorProcessing::AddThirdPartLutFile(LPCTSTR fileName);

description :
Insert a third part support LUT file to LUT poroceesing pipline

parameters :
filename[in]the file name to add
return value :
True if the function is successful or *false* if the function failed.

RemoveThirdPartLutFile

declaration :
bool HimageColorProcessing::RemoveThirdPartLutFile(LPCTSTR fileName);

description :
Remove the third part support LUT file from LUT poroceesing pipline

parameters :
filename[in]the file name to remove
return value :
True if the function is successful or *false* if the function failed.

Get1DPredefinedLUTCount

declaration :
static void HImageColorProcessing::Get1DPredefinedLUTCount()

description :
Ge current support predefined 1D LUT format count.

parameters :

return value :
numbers of predefined 1D LUT format

Get1DPredfinedLUT

declaration :
static void HimageColorProcessing::Get1DPredfinedLUT(int index, LPTSTR name,int& len);

description :
Get support predefined 1D LUT format infomation.

parameters :
index [in] the index of file format to query
name[out] name for for pedfined LUTformat
len[out] the string size of name

return value :
True if the function is successful or *false* if the function failed.

Load1DPredefinedLUT

declaration :
bool HimageColorProcessing::Load1DPredefinedLUT(int index);

description :
Set support predefined 1D LUT format infomation.

parameters :
index [in] the index of file format to query

return value :
True if the function is successful or *false* if the function failed.

HRotation

This module can be used to perform image rotation. The rotation is performed via a multi-threaded algorithm, optimized for Intel based architecture processors. Supported image formats are: Monochrome, raw Bayer, RGB, BGR and BRGX as well as YUV444, for all bit depths: 8, 10, 12,14 and 16 bit.

The destination image can be internally allocated if not provided. The destination image format will be maintained. Size in X and Y for the destination image must be specified at the calling time.

The Calculation can be performed according to 3 different algorithms: Use nearest pixel (fastest), use a bilinear interpolation, use a bicubic interpolation (slowest). It is also possible to enable an edge smoothing mode, where edges will be better maintained in the rotated image.

Special rotation case: When the center of rotation is set to [0,0] and the destination image X and Y size are swapped compared to the source image, a +/-90 degrees rotation is performed. The orientation of the rotation is defined by the angle parameter. The algorithm choice, as well as edge smoothing parameters, are ignored. Both, image image size in X and y mult be multiple of 4. If this is not the case, the operation will be performed on the closest possible size, while remaining lines or pixels will be zeroed in the destination image.

Pixels in the destination image that do not have any corresponding pixels in the source image are zeroed. Rotated pixels that do not fit in the destination image are clipped.

SetDestinationImageWidth

declaration :
void HRotation::SetDestinationImageWidth(unsigned int width)

description :
Set destination image size width in pixel.

parameters :
width [in] the destination image width size in pixel

return value :
None.

GetDestinationImageWidth

declaration :
unsigned int Hrotation::GetDestinationImageWidth()

description :
Return destination image size width in pixel.

parameters :
None.

return value :
Return destination image size width in pixel.

SetDestinationImageHeight

declaration :
void HRotation::SetDestinationImageHeight(unsigned int width)

description :
Set destination image size height in pixel.

parameters :
width [in] the destination image height size in pixel

return value :
None.

GetDestinationImageHeight

declaration :
unsigned int Hrotation::GetDestinationImageHeight()

description :
Return destination image size height in pixel.

parameters :
None

return value :
Return destination image size height in pixel.

SetCenterCoordinateX

declaration :
void HRotation::SetCenterCoordinateX(int value)

description :
Set the center point position in X for the rotation in pixel. Make sure to set the center to 0 for 90 and -90 degrees rotation. Center position can be set outside of image boundaries.

parameters :
value[in] the x position of the center of rotation

return value :
None.

GetCenterCoordinateX

declaration :
int Hrotation::GetCenterCoordinateX()

description :
Return the position in X for the center of rotation, in pixel.

parameters :
None.

return value :
Return the position in X for the center of rotation, in pixel.

SetCenterCoordinateY

declaration :
void HRotation::SetCenterCoordinateY(int value)

description :
Set the center point position in Y for the rotation in pixel. Make sure to set the center to 0 for 90 and -90 degrees rotation. Center position can be set outside of image boundaries.

parameters :
value[in] the Y position of the center of rotation

return value :
None.

GetCenterCoordinateY

declaration :
int Hrotation::GetCenterCoordinateY()

description :
Return the position in Y for the center of rotation, in pixel.

parameters :
None.

return value :
Return the position in Y for the center of rotation, in pixel.

SetRotateAngle

declaration :

void HRotation::SetRotateAngle (double angle)

description :
Set the angle of rotation in degrees. The angle must be specified in degrees, between 0 to +/-360.

parameters :
angle[in] the angle rotation in degrees.

return value :
None.

GetRotateAngle

declaration :
double Hrotation::GetRotateAngle()

description :
Return the rotation angle in degrees.

parameters :
None.

return value :
Return the rotation angle in degrees.

SetInterpolationMode

declaration :
void HRotation::SetInterpolationMode (eHRotationInterpolationMode mode)

description :
Define the pixel interpolation mode used during the calculation. Three interpolation mode can be used:
•`H_ROTATION_INTER_NN`: use nearest pixel (default).
•`H_ROTATION_INTER_LINEAR`: use bilinear interpolation.
•`H_ROTATION_INTER_CUBIC`: use bicubic interpolation.

parameters :
mode [in] the interpolation mode

return value :
None.

GetInterpolationMode

declaration :
eHRotationInterpolationMode Hrotation::GetInterpolationMode()

description :
Return the current interpolation mode

parameters :
None.

return value :
Return the interpolation mode.

SetSmoothEdge

declaration :
void HRotation::SetSmoothEdge (bool smooth)

description :
Enable or disable edge smoothing while performing the rotation. Edge Smoothing is more CPU demanding.
parameters :
smooth [in] the smooth mode

return value :
None.

GetSmoothEdge

declaration :
bool Hrotation::GetSmoothEdge()

description :
Return the current smooth mode: false = off, true = on.

parameters :
None.

return value :
Return the smooth mode as a boolean.

Rotate

declaration :
bool HRotation::Rotate(HImage* srcImage, Himage* destImage)

description :
Perform the image rotation from srcImage to destImage

parameters :

srcImage [in]: source image to be rotated
destImage [out]: destination image

return value :
Return true if the rotation was successful, false otherwise. Possible reason why the rotation failed is because the image format is not supported.

HImageResize

Changes an image size using different interpolation algorithms.

SetSourceRoi

declaration :
void HImageResize::SetSourceRoi(int offsetX, int offsetY, int sizeX, int sizeY)

void ImageResize::SetSourceRoi(int offsetX, int offsetY, int sizeX, int sizeY)

description :
Allows to set the region of interest on the source image.

parameters :
offsetX [in] The horizontal offset of the region.
offsetY [in] The vertical offset of the region.
sizeX [in] The width of the region.
sizeY [in] The height of the region.

return value :
None.

GetSourceRoi

declaration :
void HImageResize::GetSourceRoi(int& offsetX, int& offsetY, int& sizeX, int& sizeY)

void ImageResize::GetSourceRoi(ref int offsetX, ref int offsetY, ref int sizeX, ref int sizeY)

description :
Get the region of interest on the source image.

parameters :
offsetX [out] The horizontal offset of the region.
offsetY [out] The vertical offset of the region.
sizeX [out] The width of the region.
sizeY [out] The height of the region.

return value :
None.

SetDestinationSize

declaration :
void HImageResize::SetDestinationSize(int sizeX, int sizeY)

void ImageResize::SetDestinationSize(int sizeX, int sizeY)

description :
Set size of the output image.

parameters :
sizeX [in] The width of the image.
sizeY [in] The height of the image.

return value :
None.

GetDestinationSize

declaration :
void HImageResize::GetDestinationSize(ref int sizeX,ref int sizeY)

void ImageResize::GetDestinationSize(ref int sizeX, ref int sizeY)

description :
Get the size of the output image.

parameters :
sizeX [out] The width of the output image.
sizeY [out] The height of the output image.

return value :
None.

SetInterpolationMode

declaration :
void HImageResize::SetInterpolationMode(eHResizeInterpolationMode mode)

void ImageResize::SetInterpolationMode(HEnums::ResizeInterpolationMode mode)

description :
This function set the interpolation mode used for image resizing.

parameters :
mode [in] The interpolation mode :
H_RESIZE_INTER_NN : nearest neighbor interpolation
H_RESIZE_INTER_LINEAR : linear interpolation
H_RESIZE_INTER_CUBIC : cubic interpolation
H_RESIZE_INTER_LANCZOS : interpolation with Lanczos window
H_RESIZE_INTER_SUPER : supersampling interpolation

return value :
None.

GetInterpolationMode

declaration :
eHResizeInterpolationMode HImageResize::GetInterpolationMode()

HEnums::ResizeInterpolationMode ImageResize::GetInterpolationMode()

description :
This function get the Interpolation mode for image resize.

Parameters :
None.

return value :
The interpolation mode in use. For the possible values see SetInterpolationMode.

CheckImageResizeFormat

declaration :
bool HImageResize::CheckImageResizeFormat(eHImageFormat format,int bitDepth)

bool ImageResize::CheckImageResizeFormat(HEnums::ImageFormat format,int bitDepth)

description :
This function can be called to check if the image format and bit depth are supported by the image resize function.

parameters :
format [in] Source image format.

bitDepth [in] Source image bit depth.

return value :
True if the image format and bit depth combination is supported. False otherwise.

Resize

declaration :
bool HImageResize::Resize(HImage* srcImage,HImage* destImage)

bool ImageResize::Resize(HermesImage srcImage,HermesImage destImage)

description :
Perform the image resize from srcImage to destImage.

parameters :
srcImage [in] Source image to be resized
destImage [out] Resized destination image

return value :
Return true if the resize was successful, false otherwise.

CheckImageReSampleFormat

declaration :
bool HImageResize::CheckImageReSampleFormat(eHImageFormat format, int bitDepth)

bool ImageResize::CheckImageReSampleFormat(HEnums::ImageFormat format, int bitDepth)

description :
This function can be called to check if the image format and bit depth are supported by the image resample function.

parameters :
format [in] Source image format to be check
bitDepth [in] Source image bit depth to be check

return value :
True if the image format and bit depth combination is supported. False otherwise.

ReSample

declaration :
bool HImageResize::ReSample(int factor,HImage* srcImage,HImage* destImage)

bool ImageResize::ReSample(int factor,HermesImage srcImage, HermesImage destImage)

description :
Perform the image sampling from srcImage to destImage.

parameters :
factor [in] sampling factor (1:X)
srcImage [in] source image to be resampled
destImage [out] resampled destination image

return value :
Return true if the resampling was successful, false otherwise.

HImageMerge

Merge 2 images together to create a larger image.

SetMergeMode

declaration :
void HImageMerge::SetMergeMode(eHMerge_Mode mode)

void ImageMerge::SetMergeMode(HEnums::Merge_Mode mode)

description :
This function set the merge mode.

parameters :
mode [in] The merge mode to use. It can be:
H_MERGE_HORZ: Merge along the horizontal axis.
H_MERGE_VERT: Merge along vertical axis.

return value :
None.

GetMergeMode

declaration :
eHMerge_Mode HImageMerge::GetMergeMode()

HEnums::Merge_Mode ImageMerge::GetMergeMode()

description :
This function retrieve the current the merge mode.

parameters :
None.

return value :
The merge mode in use. For the possible values see SetMergeMode.

ForceOutputBRG24

declaration :
void HImageMerge::ForceOutputBRG24(bool bgr)

void ImageMerge::ForceOutputBRG24(bool bgr)

description :
This function force image merge output as BGR24 format.

parameters :
bgr [in] true to force output to BGR24 format. False to keep the source format.

return value :
None.

CheckImageMergeFormat

declaration :
bool HImageMerge::CheckImageMergeFormat(eHImageFormat format1, int bitDepth1,
eHImageFormat format2, int bitDepth2)

bool ImageMerge::CheckImageMergeFormat(HEnums::ImageFormat format1, int bitDepth1,
HEnums::ImageFormat format2, int bitDepth2)

description :
This function can be called to check if the image format and bit depth are supported by the merge function.

parameters :
format1 [in] First source image format to be check.
bitDepth1 [in] First source image bit depth to be check.
format2 [in] Second source image format to be check.
bitDepth2 [in] Second source image bit depth to be check.

return value :
True if the image format and bit depth combination is supported. False otherwise.

Merge

declaration :
bool HimageMerge::Merge(HImage* srcImage, HImage* srcImage2, HImage* destImage);

bool ImageMerge::Merge(HermesImage srcImage, HermesImage srcImage2, HermesImage destImage)

description :
Merge srcImage1 and srcImage2 to destImage.

parameters :
srcImage1 [in] First source image to be merged.
srcImage2 [in] Second source image to be merged.
destImage [out] Destination image

return value :
Return true if the merge was successful, false otherwise.

HImageMirror

This module can be used to perform image mirroring. The mirroring is performed via a multi-threaded algorithm, optimized for Intel based architecture processors. Supported image formats are: monochrome, raw Bayer, RGB, BGR, BRGX,BGR Planner as well as YUV444, for all bit depth: 8, 10, 12,14 and 16 bit. UVY422 and YUV422 format are supported with 16bit pixel depth only.

SetMirrorMode

declaration :
void HImageMirror::SetMirrorMode(eHMirror_Mode mode)

void ImageMirror::SetMirrorMode(HEnums::MirrorMode mode)

description :
This function set the mirror mode.

parameters :
mode [in] The mirror mode to use. It can be:
H_MIRROR_HORZ: Mirror via horizontal axis;
H_MIRROR_VERT: Mirror via vertical axis;
H_MIRROR_BOTH: Mirror via horizontal and vertical axis;

return value :
None.

GetMirrorMode

declaration :
eHMirror_Mode HImageMirror::GetMirrorMode()

HEnums::MirrorMode ImageMirror::GetMirrorMode()

description :
This function returns the current mirror mode.

parameters :
None.

return value :
The mirror mode in use. For the possible values see SetMirrorMode.

CheckImageMirrorFormat

declaration :
bool HImageMirror::CheckImageMirrorFormat(eHImageFormat format,int bitDepth)

bool ImageMirror::CheckImageMirrorFormat(HEnums::ImageFormat format,int bitDepth)

description :
This function can be call to check if the image format and bit depth are supported by the mirror function.

Parameters :
format [in] Source image format to be check

bitDepth [in] Source image bit depth to be check

return value :
True if the image format and bit depth combination is supported. False otherwise.

Mirror

declaration :
bool HImageMirror::Mirror(HImage* srcImage,HImage* destImage)

bool ImageMirror::Mirror(HermesImage srcImage,HermesImage destImage)

description :
Perform the image mirror from srcImage to destImage.

parameters :
srcImage [in]: source image to be mirrored
destImage [out]: destination image

return value :
Return true if the mirror was successful, false otherwise.

HBitmapOverlay

This module can used to overlay a bitmap to an image. The pixels of the original image are replaced (burned) by the one specified in the bitmap file. The bitmap file can be of any size. If the size is larger than the image, bitmap will be clipped. Bitmap format can be of any type: 8 bit, 24 bit, monochrome or color. One can use either white or black color as a transparency: All pixels set with the transparency color in the bitmap file will not be burned to the returned image.

SetBitmapFile

declaration :
void HBitmapOverlay::SetBitmapFile(LPCTSTR fileName)

void BitmapOverlay::SetBitmapFile(string fileName)

description :
This function set the bitmap file that is burned on the image

parameters :
fileName [in] the bitmap file path.

return value : none

GetBitmapFile

declaration :
LPCTSTR HBitmapOverlay::GetBitmapFile()

string BitmapOverlay::GetBitmapFile()

description :
This function returns the current bitmap file that is burned on the image

Parameters :

return value :
The bitmap file path

SetTransparencyColor

declaration :
void HbitmapOverlay::SetTransparencyColor(bool white)

void BitmapOverlay::SetTransparencyColor(bool white)

description :
This function set Transparency color. Only 2 transparency color can be defined: black or white.

Parameters :
white[IN]: transparency color is white if true, else transparency color is black.

return value : none

SetDisplayColorForBinaryBitmap

declaration :
bool HbitmapOverlay::SetDisplayColorForBinaryBitmap(int color)

bool BitmapOverlay::SetDisplayColorForBinaryBitmap(int color)

description :
Allows selection of a specific color rather than only black or white. When used with a monochrome image, the custom color will actually represent the intensity of the red channel in the selected color plan, displayed in grey level.

parameters :
color [in]: the color to be selected.

return value :
True.

IsBinaryBitmap

declaration :
bool HbitmapOverlay::IsBinaryBitmap(bool& bin)

bool BitmapOverlay::IsForBinaryBitmap(bool& bin)

description :
Check if the import bitmap file is binary bitmap.

parameters :
bin [out]: True if bitmap file is Binary Bitmap.

return value :
True if success.

SetOverlayPosition

declaration :
bool HbitmapOverlay::SetOverlayPosition(int x, int y)

bool BitmapOverlay::SetOverlayPosition(int x, int y)

description :
Set the top left position for the first pixel of the bitmap file to be overlay on the image. Default (0,0).

parameters :
x [in]: the Overlay position in X
x [in]: the Overlay position in Y

return value :
True.

SuggestCenterPosition

declaration :
bool HbitmapOverlay::SuggestCenterPosition(int imageSizeX, int imageSizeY, int& overlayPositionX, int& overlayPositionY)

bool BitmapOverlay::SuggestCenterPosition(int imageSizeX, int imageSizeY, int& overlayPositionX, int& overlayPositionY)

description :
Calculate and return the best X,Y position so that the bitmap is horizontally centered on the image. Return values can be use by SetOverlayPosition().

parameters :
imageSizeX [in]: the current destination image size X
imageSizeY [in]: the current destination image size Y
overlayPositionX[out]: suggested overlay Position X
overlayPositionX[out]: suggested overlay Position Y

return value :
True.

SuggestBottomRightPosition

declaration :
bool HbitmapOverlay::SuggestBottomRightPosition(int imageSizeX, int imageSizeY, int& overlayPositionX, int& overlayPositionY)

bool BitmapOverlay::SuggestBottomRightPosition(int imageSizeX, int imageSizeY, int& overlayPositionX, int& overlayPositionY)

description :
Calculate and return the best X,Y position so that the bitmap is vertically centered on the image. Return values can be use by SetOverlayPosition().

parameters :
imageSizeX [in]: the current destination image size X
imageSizeY [in]: the current destination image size Y
overlayPositionX[out]: suggested overlay Position X
overlayPositionX[out]: suggested overlay Position Y

return value :
True.

WriteBitmapOverlay

declaration :
bool HbitmapOverlay::WriteBitmapOverlay(HImage* image)

bool BitmapOverlay::WriteBitmapOverlay(HermesImage image)

description :
Write the currently selected bitmap to the image (perform the burning).

parameters :
image [in]: the image to write bitmap overlay in.
return value :
Return true if it was successful, false otherwise. Possible reason why the operation failed is because the image format is not supported. Contact Norpix to get the image format added.

HImageOverlay

This class can be used to add overlays on images.

SetOverlayColor

declaration :
bool HImageOverlay::SetOverlayColor(COLORREF color)

bool ImageOverlay::SetOverlayColor(int color)

description :
This function set the color used to draw the overlays.

parameters :
color [in] The color to use, in the standard windows COLORREF format.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetOverlayColor

declaration :
COLORREF HImageOverlay::GetOverlayColor()

int ImageOverlay::GetOverlayColor()

description :
Retrieve the color used to draw the overlays.

parameters :
None.

return value :
The color used, in the standard windows COLORREF format.

SetTextBackgroundColor

declaration :
bool HImageOverlay::SetTextBackgroundColor(COLORREF color)

bool ImageOverlay::SetTextBackgroundColor(int color)

description :

Set the background color of the text overlays. If both the color and background color are the same, background will be transparent.

parameters :
color [in] The color to use, in the standard windows COLORREF format.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetTextBackgroundColor

declaration :
COLORREF HImageOverlay::GetTextBackgroundColor()

int ImageOverlay::GetTextBackgroundColor()

description :
Retrieve the color used in the background of text overlays.

parameters :
param [in] param desc

return value :
The color used, in the standard windows COLORREF format.

SetAlpha

declaration :
bool HImageOverlay::SetAlpha(int alpha)

bool ImageOverlay::SetAlpha(int alpha)

description :
Set the alpha channel value (transparency) of the overlays.

parameters :
alpha [in] The alpha value from 0:Full transparency to 100:Opaque.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAlpha

declaration :
int HImageOverlay::GetAlpha()

int ImageOverlay::GetAlpha()

description :
Retrieve the alpha channel value used to draw the overlays.

parameters :
None.

return value :
The alpha value from 0:Full transparency to 100:Opaque.

SetLineThickness

declaration :
int HImageOverlay::SetLineThickness(unsigned int thickness)

bool ImageOverlay::SetLineThickness(unsigned int thickness)

description :
Set the line thickness (in pixels) for the functions that draw lines and shapes.

parameters :
thickness [int] The line thickness, in pixels.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetTextOverlayAlign

declaration :
bool HImageOverlay::SetTextOverlayAlign(int horizontalAlign = -1, int verticalAlign = -1)

bool ImageOverlay::SetTextOverlayAlign(int horizontalAlign, int verticalAlign)

description :
Set the alignment of the text overlays on the image.
//horizontalAlign
//verticalAlign

parameters :
horizontalAlign [in] <0 : left aligned >0 : right aligned =0 : centered
verticalAlign [in] <0 : top aligned >0 : bottom aligned =0 : centered

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call

GetLastError to retrieve the error code.

SetTextOverlayFont

declaration :
bool HImageOverlay::SetTextOverlayFont(LPCTSTR fontName)

bool ImageOverlay::SetTextOverlayFont(String^ fontName)

description :
Set the font used for the text overlay. The font must be installed on the system.

parameters :
fontName [in] The name of the font to use.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetTextOverlaySize

declaration :
bool HImageOverlay::SetTextOverlaySize(int fontSize)

bool ImageOverlay::SetTextOverlaySize(int fontSize)

description :
Set the font size for text overlays.

parameters :
fontSize [in] the font size in pts. Must be between [8, 72]

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteAlignedTextOverlay

declaration :
bool HImageOverlay::WriteAlignedTextOverlay(LPCTSTR text, HImage* image)

bool ImageOverlay::WriteAlignedTextOverlay(String^ text, HermesImage^ image)

description :
Write a text overlay on an image using the alignment set by SetTextOverlayAlign().

parameters :

text [in] The text string to burn on the image
image [in] The target image on which to burn the text overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteTextOverlay

declaration :
bool HImageOverlay::WriteTextOverlay(LPCTSTR text, POINT dest, HImage* image)

bool ImageOverlay::WriteTextOverlay(String^ text, Point dest, HermesImage^ image)

description :
Write a text overlay on an image at the given position.

parameters :
text [in] The text string to burn on the image
dest [in] The pixel coordinates on which the text is left aligned.
image [in] The target image on which to burn the text overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteLineOverlay

declaration :
bool HImageOverlay::WriteLineOverlay(POINT origin, POINT dest, HImage* image)

bool ImageOverlay::WriteLineOverlay(Point origin, Point dest, HermesImage^ image)

description :
Draw a line overlay on an image.

parameters :
origin [in] The coordinates of the origin pixel.
dest [in] The coordinates of the destination pixel.
image [in] The target image on which to burn the overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteArrowOverlay

declaration :
bool HImageOverlay::WriteArrowOverlay(POINT origin, POINT dest, HImage* image)

bool ImageOverlay::WriteArrowOverlay(Point origin, Point dest, HermesImage^ image)

description :
Draw an arrow overlay on an image.

parameters :
origin [in] The coordinates of the origin pixel.
dest [in] The coordinates of the destination pixel. The arrow will point here.
image [in] The target image on which to burn the overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteRectOverlay

declaration :
bool HImageOverlay::WriteRectOverlay(RECT rect, HImage* image)

bool ImageOverlay::WriteRectOverlay(Rect rect, HermesImage^ image)

description :
Draw a rectangle frame overlay on an image.

parameters :
rect [in] A RECT structure to pass the pixels coordinates that define the boundaries.
image [in] The target image on which to burn the overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteFilledRectOverlay

declaration :
bool HImageOverlay::WriteFilledRectOverlay(RECT rect, HImage* image)

bool ImageOverlay::WriteFilledRectOverlay(Rect rect, HermesImage^ image)

description :
Draw a filled rectangle overlay on an image.

parameters :
rect [in] A RECT structure to pass the pixels coordinates that define the boundaries.
image [in] The target image on which to burn the overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteEllipseOverlay

declaration :
bool HImageOverlay::WriteEllipseOverlay(RECT rect, HImage* image)

bool ImageOverlay::WriteEllipseOverlay(Rect rect, HermesImage^ image)

description :
Draw an ellipse overlay on an image.

parameters :
rect [in] A RECT structure to pass the pixels coordinates that define the horizontal and vertical boundaries.
image [in] The target image on which to burn the overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteFilledEllipseOverlay

declaration :
bool HImageOverlay::WriteFilledEllipseOverlay(RECT rect, HImage* image)

bool ImageOverlay::WriteFilledEllipseOverlay(Rect rect, HermesImage^ image)

description :
Draw a filled ellipse overlay on an image.

parameters :
rect [in] A RECT structure to pass the pixels coordinates that define the horizontal and vertical boundaries.
image [in] The target image on which to burn the overlay

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteCircleOverlay

declaration :
bool HImageOverlay::WriteCircleOverlay(POINT center, int radius, HImage* image)

bool ImageOverlay::WriteCircleOverlay(Point center, int radius, HermesImage^ image)

description :
Draw a circle overlay on an image.

parameters :
center [in] The coordinates of the pixel at the center.
radius [in] The radius (in pixel) of the circle.
image [in] The target image on which to burn the overlay.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WritePolyOverlay

declaration :
bool HImageOverlay::WritePolyOverlay(POINT pointsArray[], unsigned int pointsCount,
HImage* image)

bool ImageOverlay::WritePolyOverlay(array<Point>^ pointsArray, unsigned int pointsCount,
HermesImage^ image)

description :
Draw a polygon overlay on an image.

parameters :
pointsArray [in] An array of points that will be linked to form the polygon.
pointsCount [in] The number of points in the pointsArray.
image [in] The target image on which to burn the overlay.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteBezierOverlay

declaration :
bool HImageOverlay::WriteBezierOverlay(POINT pointsArray[], unsigned int pointsCount,
HImage* image)

bool ImageOverlay::WriteBezierOverlay(array<Point>^ pointsArray, unsigned int pointsCount,
HermesImage^ image)

description :
Same as the WritePolyOverlay function except that point are conected with bezier lines instead of straight lines.

parameters :
pointsArray [in] An array of points that will be linked to form the bezier overlay.
pointsCount [in] The number of points in the pointsArray.
image [in] The target image on which to burn the overlay.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

WriteCustomOverlay

declaration :
bool HImageOverlay::WriteCustomOverlay(POINT pointsArray[], unsigned int pointsCount,
HImage* image)

bool ImageOverlay::WriteCustomOverlay(array<Point>^ pointsArray, unsigned int pointsCount,
HermesImage^ image)

description :
Draw a custom overlay on an image. Every pixels in the pointsArray will be colored using the overlay color. The pixels are not automatically linked to each other.

parameters :
pointsArray [in] An array of points that will make the overlay.
pointsCount [in] The number of points in the pointsArray.
image [in] The target image on which to burn the overlay.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

HAudioData

Note : The audio related classes are still in active development and should be considered as betas.

This class can be used to control audio recording to an audio file.

The *HAudioData* is an audio container. It comprises the data of a single audio block and offers several functions to query the audio format.

CreateAudioData

declaration :
static HAudioData* HAudioData ::CreateAudioData();

description :
CreateHimage creates an empty HaudioData set. When it is no longer needed, free it by using *DeleteAudioData*.

parameters :
None.

return value :
A pointer to a newly created AudioData set.

CloneAudioData

declaration :
static HAudioData* HAudioData ::CloneAudioData(HAudioData* srcAudioData);

description :
This function creates a perfect copy of a source HaudioData set. When it is no longer needed, free it by using *DeleteAudioData*.

parameters :
srcAudioData [in] The source audio data to clone.

return value :
A pointer to a newly created AudioData.

CloneAudioData

declaration :
static HAudioData* HaudioData::CloneAudioData(HAudioData* srcAudioData, HAudioData* destAudioData);

description :
This function creates a perfect copy of the srcAudioData set in the destAudioData. For the

destAudioData set, only use HAudioData you created using HAudioData::CreateAudioData()

parameters :
srcAudioData [in] The audio data set source to clone.

destAudioData [in] The destination audio data set.

return value :
None.

GetSampleCount

declaration :
int HAudioData ::GetSampleCount();

description :
Return the number of audio sample in the audio data set.

parameters :
None

return value :
number of sample in current audio data set.

GetChannelNum

declaration :
int HAudioData ::GetChannelNum();

description :
Return the number of channels in the audio data set.

parameters :
None

return value :
Channels of audio data

GetAudioDataBuffer

declaration :
BYTE* HAudioData ::GetAudioDataBuffer();

description :
Return a pointer to the current audio data set.

parameters :
None

return value :
The pointer to an audio data buffer

GetFormat

declaration :
eHAudioFormat HAudioData::GetFormat()

description :
Retrieves the format of the audio data set. Possible formats are:

- `H_AUDIO_UNKNOWN`
- `H_AUDIO_INT_MSB_PLANER_PCM`
- `H_AUDIO_INT_PLANER_PCM`
- `H_AUDIO_FLOAT_MSB_PLANER_PCM`
- `H_AUDIO_FLOAT_PLANER_PCM`
- `H_AUDIO_INT_PACKED_PCM`
- `H_AUDIO_UINT_PACKED_PCM`
- `H_AUDIO_FLOAT_PACKED_PCM`

parameters :
None.

return value :
The function will return the format of the audio data set.

GetAudioBitDepth

declaration :
int HAudioData::GetAudioBitDepth()

description :
Retrieves the bit depth of the audio data set

parameters :
None.

return value :
The function will return the bit depth of the audio data set. If the a*udio data* set is empty, the returned value is 0.

GetAudioBitDepthReal

declaration :
int HAudioData::GetAudioBitDepthReal()

description :
Retrieves the real bit depth of the audio data set

parameters :
None.

return value :
The function will return the real bit depth of the audio data set. If the a*udio data* set is empty, the returned value is 0.

ReallocateAudio

declaration :
void HaudioData::ReallocateAudio(eHAudioFormat format, int channels, int samples, long audioDepth, long audioDepthReal)

description :
ReallocateAudio will delete the audio data set currently stored in the HAudioData and will reallocate memory to fit the new audio according the specified parameters. Only user created HAudioData *set* should be reallocated.

parameters:
format [in] The audio format..
channels [in] The channels of audio.
samples [in] The samples of audio data .
audioDepth [in] The real bit depth of the Audio data.
audioDepthReal [in] real bit depth of the Audio data..

return value :
none.

GetTimestamp

declaration :
long HAudioData::GetTimestamp()

description :
GetTimestamp retrieves the time stamp of the Audiodata set. The time stamp indicates the exact time at which the audio data set was captured. More precision can be obtained by calling *GetTimestampMS* to get the milliseconds part.

parameters :
None.

return value :
The absolute time stamp in seconds.

GetTimestampMS

declaration :
unsigned short HAudioData::GetTimestampMS()

description :
This will retrieve the milliseconds part of the absolute image time stamp. Depending on the latency of the operating system, the precision might be off by 1 to 3 milliseconds.

parameters :
None.

return value :
The milliseconds part of the image time stamp.

GetTimestampUS

declaration :
unsigned shot HAudioData::GetTimestampUS()

description :
The microseconds part of the absolute image time stamp. This value will be 0 unless the frame was time stamped at while using an external high performance timer device in conjunction with the image grabber.

parameters :
None.

return value :
The microsecond part of the image time stamp.

SetTimestamp

declaration :
void HAudioData::SetTimestamp(long timeStamp, unsigned short timeStampMS, unsigned short timeStampUS)

description :
This function change the time stamp of a HAudioData.

parameters :
timeStamp [in] The absolute time stamp in seconds.

timeStampMS [in] The milliseconds part of the image time stamp.

timeStampUS [in] The microsecond part of the image time stamp.

return value :
None.

HAudioConverter

AudioConverter

This module can be used to convert any supported audio format to the following audio formats:

- Unsigned integer packed 8 bits, signed integer packed 16 bit, which are required by DirectX Audio driver.
- Signed integer packed planar 32 bit, single float packed planner, which are required by ASIO driver.

ConvertToUInt8Packed

declaration :
bool HAudioConverter::ConvertToUInt8Packed(HAudioData* audioIn, HAudioData* audioOut, int mapLen = 0, int* lineMap = NULL);

description :
Convert audio data set to unsigned integer packed 8 bits. Note: value for mute level is 127.

parameters :
audioIn [in] A pointer to input audio data source.
audioOut [out] A pointer to output audio data.
mapLen [in] The number of items in “maps”.
maps [in] An array of channels to convert from. For example, if you want to convert from channels 1,3 & 4, the maps array would be [1,3,4].if *maps is NULL means channels map is same as input*

return value
True if the function is successful or *false* if the function failed.

ConvertToInt16Packed

declaration :
bool HAudioConverter::ConvertToInt16Packed(HAudioData* audioIn, HAudioData* audioOut, int mapLen = 0, int* lineMap = NULL);

description :
Convert audio data to signed integer packed 16 bits. Note: value for mute level is 0.

parameters :
See ConvertToUInt8Packed

return value
See ConvertToUInt8Packed

ConvertToInt32Planner

declaration :
`bool` HAudioConverter::ConvertToInt32Planner(HAudioData* audioIn, HAudioData* audioOut, int mapLen = 0, int* lineMap = NULL);

description :
Convert audio data to signed integer planar 32 bits. Note: value for mute level is 0.

parameters :
See ConvertToUInt8Packed

return value
See ConvertToUInt8Packed

ConvertToFloat32Planner

declaration :
`bool` HAudioConverter::ConvertToFloat32Planner(HAudioData* audioIn, HAudioData* audioOut, int mapLen = 0, int* lineMap = NULL);

description :
Convert audio data to signed single float planar format. value for mute level is 0.0

parameters :
See ConvertToUInt8Packed

return value
See ConvertToUInt8Packed

HAudioDriverDirectX

The Hermes AudioDriverDirectX module purpose is to connect to a DirectX compatible audio source via the DirectX (DirectSound) driver.

GetDriverName

declaration :
bool HAudioDriverDirectX::GetDriverName(LPTSTR name);

description :
Retrieves the audio source device and driver name.

parameters :
name [out] the name of the driver.

return value
True if the function is successful or false if the function failed. In case of failure, the driver DLL missing.

AudioInputPresent

declaration :
bool HAudioDriverDirectX::AudioInputPresent();

description :
This function checks if the audio device is available and connected. Make sure the device is present before calling any other functions in this class.

parameters :
None

return value
True if input can be used.

GetAudioInputDeviceCount

declaration :
int HAudioDriverDirectX::GetAudioInputDeviceCount()

description :
This function retrieves the number of input audio devices.

parameters :
None.

return value
Input device count.

GetAudioInputDeviceName

declaration :
bool HAudioDriverDirectX::GetAudioInputDeviceName(int index,LPTSTR name)

description :
This function retrieves the name of the audio device.

parameters:
index[in] the device index to query
name [in,out] The function will write the name of the audio device here. Allocate TCHAR [MAX_PATH].

return value :
True if the function is successful or *false* if the function failed.

OpenAudioInputDevice

declaration :
int HAudioDriverDirectX::OpenAudioInputDevice(int index)

description :
This function try to open an input audio device.

parameters:
index[in] the device index to open

return value :
Return Handle if the function is successful or *-1* if the function failed.

CloseAudioInputDevice

declaration :
bool HAudioDriverDirectX::CloseAudioInputDevice(int handle)

description :
This function try to close an input audio device.

parameters:
handle[in] the device handle from OpenAudioInputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioInputFormatCount

declaration :
int HAudioDriverDirectX::GetAudioInputFormatCount(int handle)

description :
This function query the various sampling format supported by audio input device.

parameters:
handle[in] the device handle from OpenAudioInputDevice()

return value :
Support Input format info count.

GetAudioInputFormatInfo

declaration :
bool HAudioDriverDirectX::GetAudioInputFormatInfo(int handle, int index, LPTSTR name, eHAudioFormat* format, int* bitsPerSample, int* channels, int* sampleRate);

description :
Retrieve the device capabilities for a specific sampling format: name, audio format, bit depth, sample rate, number of channels

parameters:
handle[in] the device handle from OpenAudioInputDevice()
index[int] the input format index to query
name [out] the sample rate name
format[out] Audio format
channels[out]channels
sampleRate[out]sample rate
return value :
True if the function is successful or *0* if the function failed.

SetAudioInputFormat

declaration:

bool HAudioDriverDirectX::SetAudioInputFormat(int handle, int index, DWORD channelMask, int samplesPerBuffer);

description :
Set the input format used for data capture.

parameters:
handle[in] the device handle from OpenAudioInputDevice()
index[in] audio format index to set
channelMask[in]Mask *of channels to capture from.*
samplesPerBuffer[in] samples for each audio data

return value :
True if the function is successful or *false* if the function failed.

GetAudioInputFormat

declaration:

bool HAudioDriverDirectX::GetAudioInputFormat(int handle, int& index, DWORD& channelMask, int& samplesPerBuffer);

description :
Get current input format settings for data capture.

parameters:
handle[in] the device handle from OpenAudioInputDevice()
index[out] audio format index to set
channelMask[out]Mask *of channels to capture from.*
samplesPerBuffer[out] samples for each audio data

return value :
True if the function is successful or *false* if the function failed.

StartAudioInputCapture

declaration:

bool HAudioDriverDirectX::StartAudioInputCapture(int handle, int& sampleRate, CAudioCaptureCallback* captureCallback, nt mapLen, int* channelMap);

description :

Start audio capture stream from audio Input device

parameters:
handle[in] the device handle from OpenAudioInputDevice()
sampleRate[in/out]The sample rate want to capture return result samplerate
captureCallback[in] Callback for Audio data capture
mapLen [in] The number of items in "maps".
maps [in] An array of channels to capture from. For example, if you want to convert from channels 1,3 & 4, the maps array would be [1,3,4].A NULL map means channels map is same as input

return value :
True if the function is successful or 0 if the function failed.

StopAudioInputCapture

declaration:

bool HAudioDriverDirectX::StopAudioInputCapture(int handle);

description :
Stop Capture audio
parameters:
handle[in] the device handle from OpenAudioInputDevice()

return value :
True if the function is successful or 0 if the function failed.

IsAudioInputCapturing

declaration:

bool HAudioDriverDirectX::IsAudioInputCapturing(int handle);

description :
Check if audio input device still capturing
parameters:
handle[in] the device handle from OpenAudioInputDevice()

return value :
True if audio input device is capturing else capture is stopped

---

AudioOutputPresent

declaration :
bool HAudioDriverDirectX::AudioOutputPresent();

description :
This function checks if an output audio device is available and connected. Make sure the device is present before calling any other functions in this class.

parameters :
None

return value
True if ouptut can be used.

GetAudioOutputDeviceCount

declaration :
int HAudioDriverDirectX::GetAudioOutputDeviceCount()

description :
This function retrieves the number of output audio devices and channels.

parameters :
None.
return value
Output device count.

GetAudioIOutputDeviceName

declaration :
bool HAudioDriverDirectX::GetAudioOutputDeviceName(int index,LPTSTR name)

description :
This function retrieves the name of the audio output device.

parameters:
index[in] the device index to query
name [in,out] The function will write the name of the audio device here. Allocate TCHAR [MAX_PATH].

return value :
True if the function is successful or *false* if the function failed.

OpenAudioOutputDevice

declaration :
int HAudioDriverDirectX::OpenAudioOutputDevice(int index)

description :
This function try to open an output audio device.

parameters:
index[in] the device index to open.

return value :
Return Handle if the function is successful or *-1* if the function failed.

CloseAudioOutputDevice

declaration :
bool HAudioDriverDirectX::CloseAudioOutputDevice(int handle)

description :
This function try to close a output audio device.

parameters:
handle[in] the device handle from OpenAudiooutputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioOutputSamplePerBuffer

declaration :
bool HAudioDriverDirectX::GetAudioOutputSamplePerBuffer(int handle)

description :
Get samples buffer size for the audio output device

parameters:
handle[in] the device handle from OpenAudiooutputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioOutputFormatCount

declaration :
int HAudioDriverDirectX::GetAudioOutputFormatCount(int handle)

description :
This function to query the number of output format supported by output device

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
Support Output fromat info count.

GetAudioOutputFormatInfo

declaration :
bool HAudioDriverDirectX::GetAudioOutputFormatInfo(int handle, int index, LPTSTR name, eHAudioFormat* format, int* bitsPerSample, int* channels, int* sampleRate);

description :
Get output format description for each available output format: name, audio format, bit depth, number of channels, sample rate.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
index[int] the input format index to query
name [out] the sample rate name
format[out] Audio format
channels[out]channels
sampleRate[out]sample rate
return value :
True if the function is successful or *0* if the function failed.

SetAudioOutputFormat

declaration:

bool HAudioDriverDirectX::SetAudioOutputFormat(int handle, int index, DWORD channelMask, int samplesPerBuffer);

description :
Set the output format used for audio playback.
parameters:
handle[in] the device handle from OpenAudioOutputDevice()
index[in] audio format index to set
channelMask[in]Mask *of channels to capture from.*
samplesPerBuffer[in] samples for each audio data

return value :
True if the function is successful or *false* if the function failed.

GetAudioOutputFormat

declaration:

bool HAudioDriverDirectX::GetAudioOutputFormat(int handle, int& index, DWORD& channelMask, int& samplesPerBuffer);

description :
Get current output format settings for auto playback.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
index[out] audio format index to set
channelMask[out]Mask *of channels to capture from.*
samplesPerBuffer[out] samples for each audio data

return value :
True if the function is successful or *false* if the function failed.

StartAudioOutputPlayback

declaration:

bool HAudioDriverDirectX::StartAudioOutputPLayback(int handle, int& sampleRate, CAudioPlaybackCallback* readCallback, nt mapLen, int* channelMap);

description :
Start playing an audio stream to an audio output device

parameters:

handle[in] the device handle from OpenAudioOutputDevice()
sampleRate[in/out]The sample rate want to capture return result samplerate
readCallback[in]Callback for audio playback
mapLen [in] The number of items in "maps".
maps [in] An array of channels to play from. For example, if you want to convert from channels 1,3 & 4, the maps array would be [1,3,4].if maps is NULL means channels map is same as input

return value :
True if the function is successful or 0 if the function failed.

StopAudioOutputPlayback

declaration:

bool HAudioDriverDirectX::StopAudioOutputPlayback(int handle);

description :
Stop playing an audio stream.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
True if the function is successful or 0 if the function failed.

IsAudioOutputPlaying

declaration:

bool HAudioDriverDirectX::IsAudioOutputPlaying(int handle);

description :
Check if audio input device is still playing.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
True if t audio output device is playing else playing is stopped.

SetOutputVolume

declaration:

bool HAudioDriverDirectX::SetOutputVolume(int handle,int percentage);

description :
Set Audio output volume level.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
percentage[in] Volume percentage to set

return value :
True if the function is successful or 0 if the function failed.

GetOutputVolume

declaration:

int HAudioDriverDirectX::GetOutputVolume(int handle);

description :
Get Audio output Volume.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
return value :
Volume percentage

HAudioDriverASIO

The Hermes AudioDriverASIO module purpose is to connect and control compatible ASIO audio devices via an ASIO driver. ASIO supports only one audio device at a time.

GetDriverName

declaration :
bool HAudiodriverASIO::GetDriverName(LPTSTR name);

description :
Retrieves the driver name.

parameters :
name [out] the name of the driver.

return value
True if the function is successful or false if the function failed. In case of failure, the driver DLL missing.

AudioInputPresent

declaration :
bool HAudiodriverASIO::AudioInputPresent();

description :
This function checks if the audio device is available and ready for connection. Make sure the device is present before calling any other functions in this class.

parameters :
None

return value
True if input can be used.

GetAudioInputDeviceCount

declaration :
int HAudiodriverASIO::GetAudioInputDeviceCount()

description :
This function retrieves the number of input audio devices.

parameters :
None.

return value
Input device count.

GetAudioInputDeviceName

declaration :
bool HAudiodriverASIO::GetAudioInputDeviceName(int index,LPTSTR name)

description :
This function retrieves the name of the audio.

parameters :
index[in] the device index to query
name [in,out] The function will write the name of the audio device here. Allocate TCHAR [MAX_PATH].

return value :
True if the function is successful or *false* if the function failed.

OpenAudioInputDevice

declaration :
int HAudiodriverASIO::OpenAudioInputDevice(int& index)

description :
This function try to open an input audio device.

parameters :
index[in/out] the device index to open. If an ASIO driver has opened, return the that device index

return value :
Return Handle if the function is successful or *-1* if the function failed.

CloseAudioInputDevice

declaration :
bool HAudiodriverASIO::CloseAudioInputDevice(int handle)

description :
This function try to close a input audio device.

parameters:
handle[in] the device handle from OpenAudioInputDevice()

return value :
True if the function is successful or *false* if the function failed.

OpenAudioInputHardwareSettingsDlg

declaration :

bool HAudiodriverASIO::OpenAudioInputHardwareSettingsDlg(int handle);

description :
Try to open Audio input device hardware settings if possible. Depending on the manufacturer driver implementation, this function call does not work at all time.

parameters :
handle[in] the device handle from OpenAudioInputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioInputChannelCount

declaration :

int HAudiodriverASIO::GetAudioInputChannelCount(int handle);

description :
Get support channels for current audio input device.

parameters :
handle[in] the device handle from OpenAudioInputDevice()

return value :
channels numbers of current device

GetAudioInputChannelInfoByChannelIndex

declaration:

bool HAudiodriverASIO::GetAudioInputChannelInfoByChannelIndex(int handle, int index, int* channelID, LPTSTR name, eHAudioFormat* format, int* depth, int* depthReal,int* sampleRate);

description :
Get channels format settings info by index
parameters:
handle[in] the device handle from OpenAudioInputDevice()
index[in] channel index to query
channelID[out]channelID
name[out]the channel name
format[out]the audio format of the channel
depth[out] the bit depth of the channel
depthReal[out] the real bit depth of the channel
sampleRate[out] The sample rate of the channel
return value :
True if the function is successful or 0 if the function failed.

StartAudioInputCapture

declaration:

bool HAudiodriverASIO::StartAudioInputCapture(int handle, int& sampleRate, CAudioCaptureCallback* captureCallback, nt mapLen, int* channelMap);

description :
Start capture audio from audio Input device
parameters:
handle[in] the device handle from OpenAudioInputDevice()
sampleRate[in/out]The sample rate want to capture return result samplerate
captureCallback[in] Callback for Audio data capture
mapLen [in] The number of items in "maps".
maps [in] An array of channels to capture from. For example, if you want to convert from channels 1,3 & 4, the maps array would be [1,3,4].if maps is NULL means channels map is same as input

return value :
True if the function is successful or 0 if the function failed.

StopAudioInputCapture

declaration:

bool HAudiodriverASIO::StopAudioInputCapture(int handle);

description :
Stop Capture audio
parameters:

handle[in] the device handle from OpenAudioInputDevice()

return value :
True if the function is successful or 0 if the function failed.

IsAudioInputCapturing

declaration:

bool HAudiodriverASIO::IsAudioInputCapturing(int handle);

description :
Check if audio input device still capturing
parameters:
handle[in] the device handle from OpenAudioInputDevice()

return value :
True if t audio input device is capturing else capturing is stopped

---

AudioOutputPresent

declaration :
bool HAudiodriverASIO::AudioOutputPresent();

description :
This function checks if the ouptut audio device is available and connected. Make sure the device is present before calling any other functions in this class.

parameters :
None

return value
True if ouptut can be used.

GetAudioOutputDeviceCount

declaration :
int HAudiodriverASIO::GetAudioOutputDeviceCount()

description :
This function retrieves the number of Output audio devices.

parameters :
None.

return value
Output device count.

GetAudioIOutputDeviceName

declaration :
bool HAudiodriverASIO::GetAudioOutputDeviceName(int index,LPTSTR name)

description :
This function retrieves the name of the audio.

parameters:
index[in] the device index to query
name [in,out] The function will write the name of the audio device here. Allocate TCHAR [MAX_PATH].

return value :
True if the function is successful or *false* if the function failed.

OpenAudioOutputDevice

declaration :
int HAudiodriverASIO::OpenAudioOutputDevice(int index)

description :
This function try to open an output audio device

parameters:
index[in] the device index to open

return value :
Return Handle if the function is successful or *-1* if the function failed.

CloseAudioOutputDevice

declaration :
bool HAudiodriverASIO::CloseAudioOutputDevice(int handle)

description :
This function try to open a output audio device

parameters:

handle[in] the device handle from OpenAudiooutputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioOutputSamplePerBuffer

declaration :
bool HAudiodriverASIO::GetAudioOutputSamplePerBuffer(int handle)

description :
Get Samples size for output device

parameters:
handle[in] the device handle from OpenAudiooutputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioOutputSampleRateInfoCount

declaration :
int HAudiodriverASIO::GetAudioOutputSampleRateInfoCount(int handle)

description :
This function to query sample rate number suppoprted by output device

parameters:
handle[in] the device handle from OpenAudiooutputDevice()

return value :
Samplerate info count.

GetAudioOutputSampleRateInfo

declaration :
bool HAudiodriverASIO::GetAudioOutputSampleRateInfo(int handle, int index, int* sampleRateID, LPTSTR name, int* value);

description :
Get sample rate info on each sample rates

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
index[int] the sample rate info index to query
int* [out] the sample rate ID
name [out] the sample rate name
int[out]the sample rate
return value :
True if the function is successful or *false* if the function failed.

OpenAudioOutputHardwareSettingsDlg

declaration:

bool HAudiodriverASIO::OpenAudioInputHardwareSettingsDlg(int handle);

description :
Try to open Audio input device hardware settings if possible

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
True if the function is successful or *false* if the function failed.

GetAudioOutputSampleRate

declaration:

int HAudiodriverASIO::GetAudioOutputSampleRate(int handle);

description :
Get current sample rate value

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
current sample rate value

GetAudioOutputChannelCount

declaration:

int HAudiodriverASIO::GetAudioOutputChannelCount(int handle);

description :
Get support channels for current audio ouptput device

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
channels numbers of current device

GetAudioOutputChannelInfoByChannelIndex

declaration:

bool HAudiodriverASIO::GetAudioOutputChannelInfoByChannelIndex(int handle, int index, int* channelID, LPTSTR name, eHAudioFormat* format, int* depth, int* depthReal);

description :
Get channels format settings info by index.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
index[in] channel index to query
channelID[out]channelID
name[out]the channel name
format[out]the audio format of the channel
depth[out] the bit depth of the channel
depthReal[out] the real bit depth of the channel

return value :
True if the function is successful or 0 if the function failed.

StartAudioOutputPlayback

declaration:

bool HAudiodriverASIO::StartAudioOutputPLayback(int handle, int& sampleRate, CAudioPlaybackCallback* readCallback, nt mapLen, int* channelMap);

description :

Start playing an audio stream from audio output device.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()
sampleRate[in/out]The sample rate want to capture return result samplerate
readCallback[in]Callback for audio playback
mapLen [in] The number of items in "maps".
maps [in] An array of channels to play from. For example, if you want to convert from channels 1,3 & 4, the maps array would be [1,3,4].if maps is NULL means channels map is same as input

return value :
True if the function is successful or 0 if the function failed.

StopAudioOutputPlayback

declaration:

bool HAudiodriverASIO::StopAudioOutputPlayback(int handle);

description :
Stop audio play back stream.

parameters:
handle[in] the device handle from OpenAudioOutputDevice()

return value :
True if the function is successful or 0 if the function failed.

IsAudioOutputPlaying

declaration:

bool HAudiodriverASIO::IsAudioOutputPlaying(int handle);

description :
Check if audio output device still playing.

parameters:
handle[in] the device handle from OpenAudioOutputDevice().

return value :
True if t audio output device is playing else playing is stopped.

CAudioCaptureCallback

This class is used by the `HAudioDriverASIO and HAudioDriverDirectX` class to notify the user of various actions occurring when audio capturing.It 's only for to customize own audio Recorder class.

OnDataReceived

declaration :

```
virtual void OnDataReceived(int handle,HAudioData* data);
```

description :
Override this function to be notified every time an audio buffer is capture.

parameters :
handle [in] The audio device handle
data [in] the audio dat buffer capture from audio device.

CAudioPlaybackCallback

This class is used by the `HAudioDriverASIO and HAudioDriverDirectX` class to notify the user of various actions occurring when audio playing.It 's only for to customize own audio Player class.

OnReadNewData

declaration :

```
virtual void OnReadNewData(int handle,HAudioData* data);
```

description :
Override this function to be notified every time an audio buffer to play.

parameters :
handle [in] The audio device handle
data [in] the audio data buffer to playing.

OnReadNewData

declaration :

```
virtual HAudioData* OnReadNewData(int handle);
```

description :
Override this function to tell device how to read audio data when needed.

parameters :
handle [in] The audio device handle

return value :
data [in] the audio data buffer to be read.

OnAudioSourceEvent

declaration :

```
virtual void OnAudioSourceEvent(int handle,eHAudioPlayEventCode code)
```

description :
Override this function to notify event when playing

parameters :
handle [in] The audio device handle
eHAudioPlayEventCode[in] The event code

HAudioRecorder

SetCallback

declaration :
void HAudioRecorder::SetCallback(CAudioRecorderCallback* callback)

description :
This function set the callback call used by the audio recorder.

parameters :
callback [in] A pointer to a CAudioRecorderCallback.

SetAudioDriver

declaration :
bool HAudioRecorder::SetAudioDriver(HAudioDriverDirectX* driver, int handle)

description :
Set the DirectX compatible audio device that will be used for audio capture. The device must have been initialized. The DirectX driver only allows single or dual channels acquisition (i.e. mono, stereo)

parameters :
driver [in] The DirectX driver to use.
handle [in] The handle to a DirectX device that is managed by the driver.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetAudioDriver

declaration :
bool HAudioRecorder::SetAudioDriver(HAudioDriverASIO* driver, int handle, int mapLen, int* maps, int sampleRate)

description :
Set the ASIO compatible audio device that will be used for audio capture. The device must have been initialized.

parameters :
driver [in] An array of points that will make the overlay.
handle [in] The handle to an ASIO device that is managed by the driver.
mapLen [in] The number of items in “maps”.

maps [in] An array of channels to capture from. For example, if you want to capture from channels 1,3 & 4, the maps array would be [1,3,4].

sampleRate [in] The acquisition rate (in samples per seconds).

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

SetAudioFile

declaration :
bool HAudioRecorder::SetAudioFile(HAudioFile* file)

description :
Set the target audio file the acquired audio buffered will be saved to. The target file must already be open.

parameters :
file [in] The target audio file.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

OpenChannels

declaration :
bool HAudioRecorder::OpenChannels()

description :
Open the audio channels, i.e. starts listening to the audio device. SetAudioDriver(...) must be called prior to this function. This doesn't start the recording itself, but it allows you to hear the audio input.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

CloseChannels

declaration :
bool HAudioRecorder::CloseChannels()

description :

Close the audio channels. This will stop the audio device from acquiring anymore audio buffers.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

AreChannelsOpen

declaration :
bool HAudioRecorder::AreChannelsOpen()

description :
Check if the audio channels are currently open.

parameters :
None.

return value :
True if the channels are open, *false* otherwise.

StartRecording

declaration :
bool HAudioRecorder::StartRecording()

description :
Start streaming the audio buffers received from the audio device to the target audio file.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

StopRecording

declaration :
bool HAudioRecorder::StopRecording()

description :
Stop audio recording to the audio file. The audio channels still remain open.

parameters :

None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsRecording

declaration :
bool HAudioRecorder::IsRecording()

description :
Check if the Audio Recorder is currently recording.

parameters :
None.

return value :
True if recording, false otherwise.

EnablePassthrough

declaration :
void HAudioRecorder::EnablePassthrough(HAudioDriverDirectX* driver, int handle)

description :
Open the audio pass-through. If the pass-through is open, when the channels are open, the device will also send audio buffer to the default audio-out. Note : On Windows 7 and later, pass-through is always enabled.

parameters :
driver [in] The DirectX driver.
handle [in] The handle to a DirectX device that is managed by the driver.

return value :
None.

DisablePassthrough

declaration :
void HAudioRecorder::DisablePassthrough()

description :
Close the audio pass-through

parameters :
None.

return value :
None.

CAudioRecorderCallback

This class is used by the HAudioRecorder class to notify the user each time an audio buffer is saved.

OnBufferSaved

declaration :
virtual void CAudioRecorderCallback::OnBufferSaved(unsigned __int64 index)

description :
Override this function to be notified everytime an audio buffer is captured from the audio device and successfully saved to an audio file.

parameters :
index [in] The index of the audio buffer in the audio file.

HAudioPlayer

Note : The audio related classes are still in active development and should be considered as betas.

This class can be used to control audio playback from an audio file.

OpenPlayer

declaration :
bool HAudioPlayer::OpenPlayer(HAudioDriverDirectX* driver, int handle, HWND window, HAudioFile* file, CAudioPlayerCallback* callbacks)

description :
Prepare the file to be played through a DirectX audio device output.

parameters :
driver [in] The DirectX audio driver to use.
handle [in] The handle to a DirectX device that is managed by the driver.
window [in] The handle of a valid window. The audio might be muted while the window doesn't have the focus.
file [in] The audio file to playback.
callbacks [in] The callback class used to notify the user of internal actions.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

OpenPlayer

declaration :
bool HAudioPlayer::OpenPlayer(HAudioDriverASIO* driver, int handle, HAudioFile* file, CAudioPlayerCallback* callbacks, int mapLen, int* maps, bool autoConfig=true)

description :
Prepare the file to be played through an ASIO audio device output(s).

parameters :
driver [in] The DirectX audio driver to use.
handle [in] The handle to an ASIO device that is managed by the driver.
file [in] The audio file to playback.
callbacks [in] The callback class used to notify the user of internal actions.
mapLen [in] The number of items in “maps”.
maps [in] An array of channels to playback to. For example, if you want to playback on channels 1,3 & 4, the maps array would be [1,3,4].
autoConfig [in] If true, the audio driver output format will be set to the same format as the file. Else, the output format must be set manually from the HAudioDriverASIO class.

return value :

True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

ClosePlayer

declaration :
void HAudioPlayer::ClosePlayer()

description :
Close the player, freeing all the allocated memory.

parameters :
None.

return value :
None.

Play

declaration :
bool HAudioPlayer::Play()

description :
Start the playback of an audio file. OpenPlayer must have been called previously.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Stop

declaration :
bool HAudioPlayer::Stop()

description :
Stop the playback of an audio file.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsPlaying

declaration :
bool HAudioPlayer::IsPlaying();

description :
Check if an audio file is currently playing.

parameters :
None.

return value :
True if an audio file is playing, false otherwise..

MoveTo

declaration :
bool HAudioPlayer::MoveTo(unsigned __int64 bufferIndex)

description :
Move to a specific buffer index in the audio file. This function won't work if called while a playback is running. Typically, call MoveTo, then Play.

parameters :
bufferIndex [in] The index of the buffer to move to.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

CAudioPlayerCallback

This class is used by the HAudioPlayer class to notify the user of various actions occurring in the player.

OnPlayBuffer

declaration :
virtual void CAudioPlayerCallback::OnPlayBuffer(HAudioData* data, unsigned __int64 index)

description :
Override this function to be notified every time an audio buffer is played from an audio file.

parameters :
data [in] The audio buffer currently being played on the audio device.
index [in] The index of the audio buffer in the audio file.

OnEndPlayback

declaration :
virtual void CAudioPlayerCallback::OnEndPlayback()

description :
Override this function to be notified when the playback of an audio file ends. This will be called when the playback stops automatically because the last buffer of the audio file was played.

parameters :
None.

HTcpServer

SetCallback

declaration :
void HTcpServer::SetCallback(HTcpCallbacks* callback, void* userPtr=NULL)

description :
Set the callback class (see HTcpCallbacks class).

parameters :
callback [in] A pointer to the user-defined callback class.
userPtr [in] A pointer to an user-defined data.

return value :
None.

OpenServer

declaration :
bool HTcpServer::OpenServer(in_addr ipAddress, unsigned short port)
bool HTcpServer::OpenServer(BYTE ip1, BYTE ip2, BYTE ip3, BYTE ip4, unsigned short port)
bool TcpServer::OpenServer(InAddr ipAddress, ushort port)
bool TcpServer::OpenServer(uint ipAddress, ushort port)
bool TcpServer::OpenServer(byte ip1, byte ip2, byte ip3, byte ip4, byte port)

description :
Open a TCP server, listening for incoming connections.

parameters :
ipAddress [in] The server's IP address. For .NET version `0x7f000001` would be `127.0.0.1`.
ip1 [in] The first byte of the server's IP address.
ip2 [in] The second byte of the server's IP address.
ip3 [in] The third byte of the server's IP address.
ip4 [in] The fourth byte of the server's IP address.
port [in] The port the server uses to listen for new connections.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :
```
unsigned short port = 1234;
unsigned long ip = MAKEIPADDRESS(192, 168, 0, 1);
in_addr ipAddress; ipAddress.s_addr = htonl(ip);
TcpServer->OpenServer(ipAddress, port);
```
or:
```
TcpServer->OpenServer(192, 168, 0, 1, port);
```

CloseServer

declaration :
bool HTcpServer::CloseServer()
bool TcpServer::CloseServer()

description :
Close the server, disconnecting all clients.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

IsServerOpen

declaration :
bool HTcpServer::IsServerOpen()
bool TcpServer::IsServerOpen()

description :
Check if the server is currently running.

parameters :
None.

return value :
True if the server is running or *false* otherwise.

CloseConnection

declaration :
bool HTcpServer::CloseConnection(SOCKET socket)
bool TcpServer::CloseConnection(uint socket)

description :
Close an existing connection.

parameters :
socket [in] A descriptor identifying the socket to close.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

CloseAllConnections

declaration :
void HTcpServer::CloseAllConnections()
void TcpServer::CloseAllConnections()

description :
Close all connections.

parameters :
None.

return value :
None.

GetConnectionCount

declaration :
int HTcpServer::GetConnectionCount()
int TcpServer::GetConnectionCount()

description :
Retrieve the total number of connections.

parameters :
None.

return value :
The total number of connections.

GetSocket

declaration :
bool HTcpServer::GetSocket(int connectionIndex, SOCKET& socket)
bool TcpServer::GetSocket(int connectionIndex, ref uint socket)

description :
Retrieve the socket of a connection identified by its index.

parameters :
connectionIndex [in] The zero-based connection index. Use HTcpServer::GetConnectionCount to determine the total number of connections.
socket [out] A parameter that retrieves the connection socket.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetFirstSocket

declaration :
SOCKET HTcpServer::GetFirstSocket()
uint TcpServer::GetFirstSocket()

description :
Retrieve the first socket in the connection list.

parameters :
None.

return value :
If successful, the function returns the first socket value in the connection list. In case of failure, the returned value is `INVALID_SOCKET`.

GetNextSocket

declaration :
SOCKET HTcpServer::GetNextSocket()
uint TcpServer::GetNextSocket()

description :
Retrieve the next socket in the connection list. You must call HTcpServer::GetFirstSocket before calling this function.

parameters :
None.

return value :
If successful, the function returns the next socket value in the connection list. In case of failure or the end of the list is reached, the returned value is `INVALID_SOCKET`.

example :
```
SOCKET s = TcpServer->GetFirstSocket();
while(s != INVALID_SOCKET)
{
     ...
     s = TcpServer->GetNextSocket();
}
```

Send

declaration :
bool HTcpServer::Send(SOCKET socket, void* data, int dataSizeBytes)
bool TcpServer::Send(uint socket, IntPtr data, int dataSizeBytes)

description :
Send data to a connected socket.

parameters :
socket [in] A descriptor identifying a connected socket.
data [in] A pointer to a buffer containing the data to be transmitted.
dataSizeBytes [in] The length, in bytes, of the data in buffer pointed to by the *data* parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Broadcast

declaration :
bool HTcpServer::Broadcast(void* data, int dataSizeBytes)
bool TcpServer::Broadcast(IntPtr data, int dataSizeBytes)

description :
Send data to all connected sockets.

parameters :
data [in] A pointer to a buffer containing the data to be transmitted.
dataSizeBytes [in] The length, in bytes, of the data in buffer pointed to by the *data* parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetAddress

declaration :
bool HTcpServer::GetAddress(in_addr& ipAddress, unsigned short& port)
bool HTcpServer::GetAddress(BYTE& ip1, BYTE& ip2, BYTE& ip3, BYTE& ip4,
unsigned short& port)
bool TcpServer::GetAddress(ref InAddr ipAddress, ref ushort port);
bool TcpServer::GetAddress(ref uint ipAddress, ref ushort port);
bool TcpServer::GetAddress(ref byte ip1, ref byte ip2, ref byte ip3, ref byte ip4, ref ushort port);

description :
Retrieve server's IP address and port.

parameters :
ipAddress [out] A parameter that receives the server's IP address.
ip1 [out] A parameter that receives the first byte of the server's IP address.
ip2 [out] A parameter that receives the second byte of the server's IP address.
ip3 [out] A parameter that receives the third byte of the server's IP address.
ip4 [out] A parameter that receives the fourth byte of the server's IP address.
port [out] A parameter that receives the communication port.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetPeerAddress

declaration :
bool HTcpServer::GetPeerAddress(SOCKET socket, in_addr& ipAddress, unsigned short& port)
bool HTcpServer::GetPeerAddress(SOCKET socket, BYTE& ip1, BYTE& ip2, BYTE& ip3, BYTE& ip4, unsigned short& port)
bool TcpServer::GetPeerAddress(uint socket, ref InAddr ipAddress, ref ushort port)
bool TcpServer::GetPeerAddress(uint socket, ref uint ipAddress, ref ushort port)
bool TcpServer::GetPeerAddress(uint socket, ref byte ip1, ref byte ip2, ref byte ip3, ref byte ip4, ref ushort port)

description :
Retrieve the peer's IP address and communication port.

parameters :
socket [in] A descriptor identifying a connected socket.
ipAddress [out] A parameter that receives the peer's IP address.
ip1 [out] A parameter that receives the first byte of the peer's IP address.
ip2 [out] A parameter that receives the second byte of the peer's IP address.
ip3 [out] A parameter that receives the third byte of the peer's IP address.
ip4 [out] A parameter that receives the fourth byte of the peer's IP address.
port [out] A parameter that receives the communication port.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

.NET Callback Mechanism (.NET only)

The following delegates are used to create the callback functions.

```
public delegate void OnClientConnectedNotificationDelegate(uint socket, int error, IntPtr userPtr)
public delegate void OnReceiveTcpDataNotificationDelegate(uint socket, IntPtr data,
                                                          uint dataSize, IntPtr userPtr)
public delegate void OnClientDisconnectedNotificationDelegate(uint socket, IntPtr userPtr)
```

parameters :
socket [out] A descriptor identifying the connection socket.
data [out] A pointer to a buffer containing the received data.
dataSize [out] The length, in bytes, of the data in buffer pointed to by the *data* parameter.
userPtr [out] The value set in the *SetCallbackPtr* function.

SetCallbackPtr

declaration :
```
void TcpServer::SetCallbackPtr(IntPtr ptr)
```

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
ptr [in] A pointer.

return value :
None.

SetCallbackOnClientConnected

declaration :
```
void TcpServer::SetCallbackOnClientConnected(OnClientConnectedNotificationDelegate func)
```

description :
This function set the callback function called when a new connection is created.

parameters :
func [in] the function delegate.

return value :
None.

SetCallbackOnReceiveData

declaration :
void TcpServer::SetCallbackOnReceiveData(OnReceiveTcpDataNotificationDelegate func)

description :
This function set the callback function called when data is received from the peer.

parameters :
func [in] the function delegate.

return value :
None.

SetCallbackOnClientDisconnected

declaration :
void TcpServer::SetCallbackOnClientDisconnected(OnClientDisconnectedNotificationDelegate func)

description :
This function set the callback function called every time a connection is closed.

parameters :
func [in] the function delegate.

return value :
None.

HTcpClient

SetCallback

declaration :
void HTcpClient::SetCallback(HTcpCallbacks* callback, void* userPtr=NULL)

description :
Set the callback class (see HTcpCallbacks class).

parameters :
callback [in] A pointer to the user-defined callback class.
userPtr [in] A pointer to an user-defined buffer.

return value :
None.

IsConnected

declaration :
bool HTcpClient::IsConnected() const
bool TcpClient::IsConnected()

description :
Check if the client is currently connected.

parameters :
None.

return value :
True if the client is connected or *false* otherwise.

ConnectTo

declaration :
bool HTcpClient::ConnectTo(in_addr localAddress, in_addr remoteAddress, unsigned short port)
bool HTcpClient::ConnectTo(BYTE localIp1, BYTE localIp2, BYTE localIp3, BYTE localIp4,
BYTE remoteIp1, BYTE remoteIp2, BYTE remoteIp3, BYTE remoteIp4, unsigned short port)
bool TcpClient::ConnectTo(InAddr localAddress, InAddr remoteAddress, ushort remotePort)
bool TcpClient::ConnectTo(uint localAddress, uint remoteAddress, ushort remotePort)
bool TcpClient::ConnectTo(byte localIp1, byte localIp2, byte localIp3, byte localIp4, byte remoteIp1,
byte remoteIp2, byte remoteIp3, byte remoteIp4, ushort port)

description :
Connect to a TCP server using the specified network interface identified by a local IP address.

parameters :
localAddress [in] The local IP address.
remoteAddress [in] The remote IP address to connect to.
localIp1 [in] The first byte of the local IP address.
localIp2 [in] The second byte of the local IP address.
localIp3 [in] The third byte of the local IP address.
localIp4 [in] The fourth byte of the local IP address.
remoteIp1 [in] The first byte of the remote IP address.
remoteIp2 [in] The second byte of the remote IP address.
remoteIp3 [in] The third byte of the remote IP address.
remoteIp4 [in] The fourth byte of the remote IP address.
port [in] The communication port.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

Disconnect

declaration :
bool HTcpClient::Disconnect()
bool TcpClient::Disconnect()

description :
Close the current connection.

parameters :
None.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetSocket

declaration :
SOCKET HTcpClient::GetSocket() const
uint TcpClient::GetSocket()

description :
Retrieve the connection socket.

parameters :
None.

return value :
If connected the function returns the existing connection socket or `INVALID_SOCKET` otherwise.

Send

declaration :
bool HTcpClient::Send(void* data, int dataSizeBytes)
bool TcpClient::Send(IntPtr data, int dataSizeBytes)

description :
Send data over a connection.

parameters :
data [in] A pointer to a buffer containing the data to be transmitted.
dataSizeBytes [in] The length, in bytes, of the data in buffer pointed to by the *data* parameter.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

GetPeerAddress

declaration :
bool HTcpClient::GetPeerAddress(in_addr& ipAddress, unsigned short& port)
bool HTcpClient::GetPeerAddress(BYTE& ip1, BYTE& ip2, BYTE& ip3, BYTE& ip4,
unsigned short& port)
bool TcpClient::GetPeerAddress(ref InAddr ipAddress, ref ushort port)
bool TcpClient::GetPeerAddress(ref uint ipAddress, ref ushort port)
bool TcpClient::GetPeerAddress(ref byte ip1, ref byte ip2, ref byte ip3, ref byte ip4, ref byte port)

description :
Retrieve the peer's IP address and communication port.

parameters :
ipAddress [out] A parameter that receives the peer's IP address.
ip1 [out] A parameter that receives the first byte of the peer's IP address.
ip2 [out] A parameter that receives the second byte of the peer's IP address.
ip3 [out] A parameter that receives the third byte of the peer's IP address.
ip4 [out] A parameter that receives the fourth byte of the peer's IP address.
port [out] A parameter that receives the communication port.

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

.NET Callback Mechanism (.NET only)

The following delegates are used to create the callback functions.

```
public delegate void OnClientConnectedNotificationDelegate(uint socket, int error, IntPtr userPtr)
public delegate void OnReceiveTcpDataNotificationDelegate(uint socket, IntPtr data,
                                                          uint dataSize, IntPtr userPtr)
public delegate void OnClientDisconnectedNotificationDelegate(uint socket, IntPtr userPtr)
```

parameters :
socket [out] A descriptor identifying the connection socket.
data [out] A pointer to a buffer containing the received data.
dataSize [out] The length, in bytes, of the data in buffer pointed to by the *data* parameter.
userPtr [out] The value set in the *SetCallbackPtr* function.

SetCallbackPtr

declaration :
```
void TcpClient::SetCallbackPtr(IntPtr ptr)
```

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
ptr [in] A pointer.

return value :
None.

SetCallbackOnClientConnected

declaration :
```
void TcpClient::SetCallbackOnClientConnected(OnClientConnectedNotificationDelegate func)
```

description :
This function set the callback function called when a new connection is created.

parameters :
func [in] the function delegate.

return value :
None.

SetCallbackOnReceiveData

declaration :
void TcpClient::SetCallbackOnReceiveData(OnReceiveTcpDataNotificationDelegate func)

description :
This function set the callback function called when data is received from the peer.

parameters :
func [in] the function delegate.

return value :
None.

SetCallbackOnClientDisconnected

declaration :
void TcpClient::SetCallbackOnClientDisconnected(OnClientDisconnectedNotificationDelegate func)

description :
This function set the callback function called every time a connection is closed.

parameters :
func [in] the function delegate.

return value :
None.

HTcpCallbacks

OnClientConnected

declaration :
void HTcpCallbacks::OnClientConnected(SOCKET socket, void* userPtr = NULL)

description :
This callback function is called when a new connection is created.

parameters :
socket [out] A descriptor identifying the connection socket.
userPtr [out] The value set in the *SetCallback* function.

return value :
None.

OnReceiveData

declaration :
void HTcpCallbacks::OnReceiveData(SOCKET socket, void* data,
unsigned int dataSize, void* userPtr = NULL)

description :
This callback function is called when data is received from peer.

parameters :
socket [out] A descriptor identifying the connection socket.
data [out] A pointer to a buffer containing the received data.
dataSize [out] The length, in bytes, of the data in buffer pointed to by the *data* parameter.
userPtr [out] The value set in the *SetCallback* function.

return value :
None.

OnClientDisconnected

declaration :
void HTcpCallbacks::OnClientDisconnected(SOCKET socket, void* userPtr = NULL)

description :
This callback function is called when a connection is closed.

parameters :
socket [out] A descriptor identifying the closing socket.
userPtr [out] The value set in the *SetCallback* function.

return value :
None.

HUdpServer

SetCallback

declaration :
void HUdpServer::SetCallback(HUdpCallbacks* callback, void* userPtr=NULL)

description :
Set the callback class (see the HUdpCallbacks class).

parameters :
callback [in] A pointer to the user-defined callback class.
userPtr [in] A pointer to an user-defined data.

return value :
None.

Open

declaration :
bool HUdpServer::Open(in_addr ip, unsigned short port, bool useHeader, int options)
bool UdpServer::Open(InAddr ip, ushort port, bool useHeader, int options)
bool UdpServer::Open(uint ip, ushort port, bool useHeader, int options)

description :
Open a UDP server.

parameters :
ip [in] The server's IP address. For .NET version `0x7f000001` would be `127.0.0.1`.
port [in] The port on which the server listens to.
useHeader [in] If *true,* enables the datagram header transmission.
options [in] A parameter identifying the socket options (for example `SO_BROADCAST`).

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :
```
unsigned short port = 1234;
unsigned long ip = MAKEIPADDRESS(192, 168, 0, 1);
in_addr ipAddr;
ipAddr.s_addr = htonl(ip);
UdpServer->Open(ipAddr, port, true, SO_BROADCAST|SO_REUSEADDR);//use header
```

Close

declaration :
void HUdpServer::Close()
void UdpServer::Close()

description :
Close the server.

parameters :
None.

return value :
None.

IsOpened

declaration :
bool HUdpServer::IsOpened() const
bool UdpServer::IsOpened()

description :
Check if the server is currently running.

parameters :
None.

return value :
True if the server is running or *false* otherwise.

SendTo

declaration :
int HUdpServer::SendTo(in_addr address, unsigned short port, const char* buffer, int bufferLen)
int UdpServer::SendTo(InAddr address, ushort port, IntPtr buffer, int bufferLen)
int UdpServer::SendTo(uint address, ushort port, IntPtr buffer, int bufferLen)

description :
Send data to a specific destination.

parameters :
address [in] The destination IP address. For .NET version `0x7f000001` would be `127.0.0.1`.
port [in] The destination port.
buffer [in] A pointer to a buffer containing the data to be transmitted.
bufferLen [in] The length, in bytes, of the data in buffer pointed to by the *buffer* parameter.

return value :
This function returns the total number of bytes sent.

SendToEx

declaration :
int HUdpServer::SendToEx(in_addr address, unsigned short port, const char* buffer, int bufferLen)
int UdpServer::SendToEx(InAddr address, ushort port, IntPtr buffer, int bufferLen)
int UdpServer::SendToEx(uint address, ushort port, IntPtr buffer, int bufferLen)

description :
Send data to a specific destination. Each datagram has a header structure, described below:

```
struct CDatagramHeader
{
     GUID guid;          //datagram ID
     UINT dataSize;      //datagram size in bytes
     UINT totalSize;     //total number of bytes to be transmitted
     UINT index;         //datagram index
     UINT count;         //total number of datagrams to be transmitted
};
```

parameters :
address [in] The destination IP address. For .NET version `0x7f000001` would be `127.0.0.1`.
port [in] The destination port.
buffer [in] A pointer to a buffer containing the data to be transmitted.
bufferLen [in] The length, in bytes, of the data in buffer pointed to by the *buffer* parameter.

return value :
This function returns the total number of bytes sent.

.NET Callback Mechanism (.NET only)

The following delegate is used to create the callback function.

```
public delegate void OnReceiveUdpDataNotificationDelegate(uint socket, IntPtr data,
                                                          uint dataSize, SockAddrIn sender, IntPtr userPtr)
```

parameters :
socket [out] A descriptor identifying the connection socket.
data [out] A pointer to a buffer containing the received data.
dataSize [out] The length, in bytes, of the data in buffer pointed to by the *data* parameter.
sender [out] A pointer to a buffer containing the address and port of the sender. May be **null.**
userPtr [out] The value set in the *SetCallbackPtr* function.

SetCallbackPtr

declaration :
```
void UdpServer::SetCallbackPtr(IntPtr ptr)
```

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
ptr [in] A pointer.

return value :
None.

SetCallbackOnReceiveData

declaration :
```
void UdpServer::SetCallbackOnReceiveData(OnReceiveUdpDataNotificationDelegate func)
```

description :
This function set the callback function called when data is received.

parameters :
func [in] the function delegate.

return value :
None.

HUdpClient

SetCallback

declaration :
void HUdpClient::SetCallback(HUdpCallbacks* callback, void* userPtr=NULL)

description :
Set the callback class (see the HUdpCallbacks class).

parameters :
callback [in] A pointer to the user-defined callback class.
userPtr [in] A pointer to an user-defined data.

return value :
None.

Connect

declaration :
bool HUdpClient::Connect(bool useHeader, int options)
bool UdpClient::Connect(bool useHeader, int options)

description :
Initiate an UDP communication.

parameters :
useHeader [in] If *true,* enables the datagram header transmission.
options [in] A parameter identifying the socket options. (e.g. `SO_BROADCAST|SO_REUSEADDR`)

return value :
True if the function is successful or *false* if the function failed. In case of failure, immediately call GetLastError to retrieve the error code.

example :
```
UdpClient->Connect(true, SO_BROADCAST);//use header and
                                        enable transmission of
                                        broadcast messages.
```

Disconnect

declaration :
void HUdpClient::Disconnect()
void UdpClient::Disconnect()

description :
Close the communication.

parameters :
None.

return value :
None.

IsConnected

declaration :
bool HUdpClient::IsConnected() const
bool UdpClient::IsConnected()

description :
Check if the UDP communication is currently active.

parameters :
None.

return value :
True if the communication is active or *false* otherwise.

SendTo

declaration :
int HUdpClient::SendTo(in_addr address, unsigned short port, const char* buffer, int bufferLen)
int UdpClient::SendTo(InAddr address, ushort port, IntPtr buffer, int bufferLen)
int UdpClient::SendTo(uint address, ushort port, IntPtr buffer, int bufferLen)

description :
Send data to a specific destination.

parameters :
address [in] The destination IP address. For .NET version `0x7f000001` would be `127.0.0.1`.
port [in] The destination port.
buffer [in] A pointer to a buffer containing the data to be transmitted.
bufferLen [in] The length, in bytes, of the data in buffer pointed to by the *buffer* parameter.

return value :
This function returns the total number of bytes sent.

SendToEx

declaration :
int HUdpClient::SendToEx(in_addr address, unsigned short port, const char* buffer, int bufferLen)
int UdpClient::SendToEx(InAddr address, ushort port, IntPtr buffer, int bufferLen)
int UdpClient::SendToEx(uint address, ushort port, IntPtr buffer, int bufferLen)

description :
Send data to a specific destination. Each datagram has a header structure, described below:

```
struct CDatagramHeader
{
     GUID guid;          //datagram ID
     UINT dataSize;      //datagram size in bytes
     UINT totalSize;     //total number of bytes to be transmitted
     UINT index;         //datagram index
     UINT count;         //total number of datagrams to be transmitted
};
```

parameters :
address [in] The destination IP address. For .NET version `0x7f000001` would be `127.0.0.1`.
port [in] The destination port.
buffer [in] A pointer to a buffer containing the data to be transmitted.
bufferLen [in] The length, in bytes, of the data in buffer pointed to by the *buffer* parameter.

return value :
This function returns the total number of bytes sent.

.NET Callback Mechanism (.NET only)

The following delegate is used to create the callback function.

```
public delegate void OnReceiveUdpDataNotificationDelegate(uint socket, IntPtr data,
                                                          uint dataSize, SockAddrIn sender, IntPtr userPtr)
```

parameters :
socket [out] A descriptor identifying the connection socket.
data [out] A pointer to a buffer containing the received data.
dataSize [out] The length, in bytes, of the data in buffer pointed to by the *data* parameter.
sender [out] A pointer to a buffer containing the address of the data sender. May be **null.**
userPtr [out] The value set in the *SetCallbackPtr* function.

SetCallbackPtr

declaration :
```
void UdpClient::SetCallbackPtr(IntPtr ptr)
```

description :
This function set the user pointer member than will be returned as a parameter of the callback functions.

parameters :
ptr [in] A pointer.

return value :
None.

SetCallbackOnReceiveData

declaration :
```
void UdpClient::SetCallbackOnReceiveData(OnReceiveUdpDataNotificationDelegate func)
```

description :
This function set the callback function called when data is received.

parameters :
func [in] the function delegate.

return value :
None.

HUdpCallbacks

OnReceiveData

declaration :

```
void HUdpCallbacks::OnReceiveData(SOCKET socket, void* data, unsigned int dataSize,
                                  const sockaddr_in* sender, void* userPtr = NULL)
```

description :
This callback function is called when data is received.

parameters :
socket [out] A descriptor identifying the connection socket.
data [out] A pointer to a buffer containing the received data.
dataSize [out] The length, in bytes, of the data in buffer pointed to by the *data* parameter.
sender [out] A pointer to a buffer containing the address and port of the sender. May be NULL.
userPtr [out] The value set in the *SetCallback* function.

return value :
None.

HNetUtils

GetIPAddresses

declaration :
bool GetIPAddresses(in_addr** addresses, int* count)

description :
Retrieve a linked list of IP addresses available on the local computer.
Note: A network adapter may use multiple IP addresses.

parameters :
addresses [out] A pointer to a buffer that contains a linked list of in_addr structures.
count [out] A pointer to a variable that specifies the count of in_addr structures.

return value :
True if successful or *false* otherwise.

GetIPAddressCount

declaration :
int GetIPAddressCount()

description :
Retrieve the number of IP addresses found on all active network adapters on the local computer.
Note: A network adapter may use multiple IP addresses.

parameters :
None.

return value :
The number of IP addresses configured on the local host.

GetIPAddressByIndex

declaration :
bool GetIPAddressByIndex(in_addr* ipAddress, int index = 0)
bool GetIPAddressByIndex(BYTE& ip1, BYTE& ip2, BYTE& ip3, BYTE& ip4, int index = 0)

description :
Retrieve the local IP address identified by its index.

parameters :
ipAddress [out] A pointer to a pre-allocated buffer that receives the requested IP address.
ip1 [out] A parameter that receives the first byte of the requested IP address.
ip2 [out] A parameter that receives the second byte of the requested IP address.
ip3 [out] A parameter that receives the third byte of the requested IP address.
ip4 [out] A parameter that receives the fourth byte of the requested IP address.
index [in] Identifies which IP address is requested. The index must be in range of 0 through GetIPAddressCount() - 1.

return value :
True if successful or *false* otherwise.

GetAdapterNameByIP

declaration :
bool GetAdapterNameByIP(int ipIndex, wchar_t* adapterName, int length)
bool GetAdapterNameByIP(in_addr ipAddress, wchar_t* adapterName, int length)

description :
Retrieve adapter name identified by an IP address on the local computer.

parameters :
ipIndex [in] The index of an IP address identifying the network adapter whose name is wished to be retrieved.
ipAddress [in] The IP address identifying the network adapter whose name is wished to be retrieved.
adapterName [in,out] A pre-allocated buffer that will receive the adapter name.
length [in] The size in wchar_t of the destination string buffer pointed to by *adapterName*.

return value :
True if successful or *false* otherwise.

GetAdapterLinkSpeed

declaration :
bool GetAdapterLinkSpeed(LPCTSTR adapterName, unsigned long& bps)

description :
Retrieve the speed of the network adapter identified by its name.

parameters :
adapterName [in] The name of the network adapter whose speed is to be retrieved.
bps [out] The speed of the network interface in bits per second.

return value :
True if successful or *false* otherwise.

GetHostName

declaration :
bool GetHostName(LPTSTR hostName, int& length)

description :
Retrieve the standard host name for the local computer.

parameters :
hostName [in,out] A pre-allocated buffer that will receive the computer name.
length [in,out] As input, it contains the size in TCHARs of the destination string buffer pointed to by *hostName*. As output, it holds the size in TCHARs of the computer name.

return value :
True if successful or *false* otherwise.

ResolveIPAddress

declaration :
bool ResolveIPAddress(LPCTSTR hostname, in_addr** addresses, int* count)

description :
Provides protocol-independent translation from a host name to an IPv4 address.

parameters :
hostname [in] A pointer to a NULL-terminated string that contains a host name.
addresses [out] A pointer to a buffer that contains a linked list of in_addr structures.
count [out] A pointer to a variable that specifies the count of in_addr structures.

return value :
True if successful or *false* otherwise.

ResolveHostName

declaration :
bool ResolveHostName(in_addr ip, LPTSTR hostname, int length)

description :
Provides protocol-independent translation from an IPv4 address to a host name.

parameters :
ip [in] The address of the host whose name is wished to be retrieved.
hostname [out] A pointer to a pre-allocated buffer that will receive the name on successful return.
length [in] The size in TCHARs of the buffer pointed to by *hostname*.

return value :
True if successful or *false* otherwise.

GetSubnetMask

declaration :
bool GetSubnetMask(const in_addr& ipAddress, in_addr& subnetmask)

description :
Retrieves the subnet mask of the network interface identified by an IPv4 address.

parameters :
ipAddress [in] The address identifying the network adapter for which the subnet mask is retrieved.
subnetmask [out] A reference to a in_addr structure that receives the subnet mask.

return value :
True if successful or *false* otherwise.

StringToIpv4Address

declaration :
in_addr StringToIpv4Address(LPCTSTR str)

description :
Converts a NULL-terminated string containing an IPv4 dotted-decimal address into an in_addr structure.

parameters :
str [in] A NULL-terminated string representing an IPv4 address expressed in the Internet standard "." (dotted) notation (i.e. “192.168.0.1”)

return value :
The converted IPv4 address as an in_addr structure.

Ipv4AddressToString

declaration :
bool Ipv4AddressToString(in_addr ipAddress, LPTSTR buffer, int length)

description :
Converts an IPv4 address into an ASCII string in Internet standard dotted-decimal format.

parameters :
ipAddress [in] An in_addr structure that represents an Internet host address.
buffer [in, out] A pointer to a pre-allocated buffer that will receive the dotted string representation.
length [in] The size in TCHARs of the buffer pointer to by *buffer*.

return value :
True if successful or *false* otherwise.

Metadada

The HMetadata hold a metadata that is attached to an image. Accessing the metadatas attached to an image is done through the functions of the HImage class. While the HMetadata class represent the generic container for the metadata, the classes derived from HMetadata are used to read/write and format the the actual data. If you want to use metadata, take some time to check the code in the various header files (.h) provided in Hermes.

MetadataID.h

This file hold the unique identifiers for all generic metadata types along with all the metadata types created by NorPix. A third party developper can create new metadata types using one of the ID in the range of [UID_METADATA_USER_FIRST, UID_METADATA_USER_LAST]

HMetadata.h

The HMetadata class is the base class used to manage a metadata. It has an unique identifer that can be used to know which HMetadata derived class can be use to decipher the metadata content. The HMtadata class itself only hold the metadata and its size, along with a string representation of the metadata (if available).

NpxMetadata.h

This file hold all the derived classes used to manage the generic and NorPix defined metadata types.

Generic metadata types

-HMetadataGenericBool - 1 byte "bool" value
-HMetadataGenericByte - 1 byte value
-HMetadataGenericShort - 2 bytes "short" value
-HMetadataGenericUShort - 2 bytes "unsigned short" value
-HMetadataGenericInt - 4 bytes "int" value
-HMetadataGenericUInt - 4 bytes "unsigned int" value
-HMetadataGenericDouble - 8 bytes "double" value
-HMetadataGenericString - a "wchar_t" null-terminated string (variable length)
-HMetadataGenericBinary - binary data (variable length)

NorPix metadata types
-HMetadataNorPixTime64 - a timestamp in time_t (64-bit) along with MS and US precision.
-HMetadataNorPixLTC - a LTC value HH:MM:SS:FF.

HMetadataInfo.h

The HMetadataInfo class holds all the details on a specific Metadata type. Among the informations provided, you can find the metadata name, a short description, its format (bool, int, string, etc), its

size and a "format string" that can be used to format the metadata to a readable string.

HMetadataManager

The Metadata Manager is an utility class that manage metadata usage on the system.

RegisterMetadataType

declaration :
bool RegisterMetadataType(HMetadataInfo* info)

description :
Register metadata information for a 3rd party metadata type.

parameters :
info [in] The metadata information.

return value :
True if the value was successfully written to the registry. False otherwise.

RegisterGenericTypes

declaration :
bool RegisterGenericTypes()

description :
Register all the generic metadata types informations.

parameters :
None.

return value :
True if the registration was successful. False otherwise.

RegisterNorPixTypes

declaration :
bool RegisterNorPixTypes()

description :
Register all the NorPix metadata types informations.

parameters :
None.

return value :
True if the registration was successful. False otherwise.

GetMetadataTypeCount

declaration :
int GetMetadataTypeCount()

description :
Retrieve the total count of metadata types registered on the system. This include any registered generic types, NorPix types and 3rd party types.

parameters :
None.

return value :
The total count of metadata types registered on the system.

GetMetadataTypeByIndex

declaration :
HMetadataInfo* GetMetadataTypeByIndex(unsigned int index)

description :
Retrieve the metadata information from the registry, for the metadata type occupying this index.

parameters :
index [in] The index, must be between [0, GetMetadataTypeCount()-1]

return value :
A pointer to the metadata info or NULL if the index is out of bounds.

GetMetadataTypeByID

declaration :
HMetadataInfo* GetMetadataTypeByID(unsigned int id)

description :
Retrieve the metadata information from the registry, for the metadata type having this unique identifier.

parameters :
id [in] The unique identifier of the metadata type.

return value :
A pointer to the metadata info or NULL if the unique ID is not registered on the system.

RebuildMetadataString

declaration :
void RebuildMetadataString(HMetadata* metadata)

description :
Rebuild the string representation of the metadata. This can be done for all generic/NorPix types. For 3rd party types, the HMetadataInfo's FormatString will be used (if available) to rebuild the string.

parameters :
metadata [in,out] The metadata structure.

return value :
None.

Sequence files

For information on the sequence file format, see the documentation accessible from the Start Menu > NorPix > Tools > Sequence Format Documentation.