OpenNI 2.0

Getting Started

Installing OpenNI SDK

Installing OpenNI SDK on Windows

Double-click the provided msi file to install the SDK on your Windows machine.

The installation performs the following:

- Copies the SDK to the target directory (default is C:\Program Files\OpenNI2 or C:\Program Files (x86)\OpenNI2)
- Installs a USB driver to be used with OpenNI-compliant devices
- Defines environment variables to be used when developing OpenNI applications (see **Visual Studio**)

Installing OpenNI SDK on Linux

1. Extract the tarball to a directory of your choice
2. Go into this directory and run the install script:
```
    ./install.sh
```

The installation creates udev rules which will allow usage of OpenNI-compliant USB devices without root priviledges.

Samples

OpenNI SDK arrives with pre-compiled samples that can be run immediately after installation.

Under the installation directory, go to the Samples/Bin directory and run any of the samples there. Note that some samples have a graphical interface and may require a more powerfull graphic accelerator.

Creating new project that uses OpenNI

Visual Studio

1. Open a new project or an existing one with which you want to use OpenNI
2. In the Visual Studio menu, open the Project menu and choose Project Properties.
3. In the C/C++ section, under the "General" node, select "Additional Include Directories" and add "$(OPENNI2_INCLUDE)" (if you use the 32-bit version) or "$(OPENNI2_INCLUDE64)" (if you use the 64-bit version). These are environment variables that point to the location of the OpenNI Include directory. (The defaults are C:\Program Files\OpenNI2\Include or C:\Program Files (x86)\OpenNI2\Include)
4. In the Linker section, under the "General" node, select "Additional Library Directories" and add "$(OPENNI2_LIB)" (if you use the 32-bit version) or "$(OPENNI2_LIB64)" (if you use the 64-bit version). These are environment variables that point to the location of the OpenNI Lib directory. (The defaults are C:\Program Files\OpenNI2\Lib or C:\Program Files (x86)\OpenNI2\Lib)
5. In the Linker section, under the input node, select "Additional Dependencies" and add OpenNI2.lib or OpenNI2.lib
6. Ensure that you add the Additional Include and Library directories to both your Release and Debug configurations.
7. Copy all the files from OpenNI's redist directory (see environment variable "$(OPENNI2_REDIST)" or "$(OPENNI2_REDIST64)") to your working directory. (The defaults are C:\Program Files\OpenNI2\Redist or C:\Program Files (x86)\OpenNI2\Redist). Be aware that when you run from command line, the working directory is the directory where the executable can be found, and where you run from Visual Studio the default directory is where the project file (.vcproj, .vcxproj) can be found.

   > **Note:**
   >
   > You may ask Visual Studio to change working directory when debugging to the directory where the executable is by

> changing "Project Properties" -> "Debugging" -> "Working Directory" to "$(TargetDir)". Note that this setting is not kept as part of the project settings, but on a per-user, per-configuration basis.

GCC / GNU Make

In the following section, refers to the directory to where OpenNI SDK was extracted. Note that the installation does not define such an environment variable. Either define it yourself or use the full path.

1. Add the SDK Include directory, $OPENNI_DIR/Include, to your include path (-I)
2. Copy the files from the Redist directory, $OPENNI_DIR/Redist, to your execution directory
3. Add the execution directory to your lib path (-L)
4. Add libOpenNI2 to your library list (-l)
5. It is highly suggested to also add the "-Wl,-rpath ./" to your linkage command. Otherwise, the runtime linker will not find the libOpenNI.so file when you run your application. (default Linux behavior is to look for shared objects only in /lib and /usr/lib).

Writing an Application

- Your code should include **OpenNI.h** header file.
- The entire C++ API is available under the openni namespace.
- Be sure to call **openni::OpenNI::initialize()**, to make sure all drivers are loaded If no drivers are found, this function will fail. If it does, you can get some basic log by calling **openni::OpenNI::getExtendedError()** (which returns a string) Note that usually this method fails because OpenNI redist files weren't copied to the working directory.
- When closing your application, call **openni::OpenNI::shutdown()**, to allow OpenNI to close properly (unload drivers and such).
- Open a device using its URI. You can get a list of available devices using **openni::OpenNI::enumerateDevices()**. Enumeration returns an array of **openni::DeviceInfo** objects, which include (among other things) the device URI. If you don't care which device

to use, you can specify **openni::ANY_DEVICE** as the URI. (to work with .oni files, use the path to the file as its URI)

- Create a video stream by specifying the device and the sensor.
- Be sure to **destroy** the stream and **close** the device when you're done.

C++ API Conventions

Classes vs. Structs

In general, classes are preferred, along with getX() / setX() methods. This allows extending the API in the future without breaking compatibility. In cases of "super-primitives" (like **openni::RGB888Pixel**), structs are used.

Return Values from Methods

- All getters return the value as their return value, and cannot fail, for example: **openni::VideoFrameRef::getWidth()**. If the value is a complex type, it will be returned as const reference, for example: **openni::Device::getDeviceInfo()**. If a specific property doesn't exist for the specific object, a meaningful value will be returned.

- stop / destroy / close methods does not have return values, for example: **openni::Device::close()**.

- All other methods return **openni::Status**, for example: **openni::VideoStream::create()**.

Arguments to Methods

- Output arguments, as well as in/out arguments are always passed as a pointer, for example: **openni::VideoStream::readFrame()**.
- Input arguments:
  - Primitives are passed by value, for example: **openni::VideoMode::setPixelFormat()**.
  - Complex types are passed as const reference, for example: **openni::VideoStream::create()**.
  - In the rare cases where the object identity matters, and not it's content, it will be passed as a pointer, for example: openni::OpenNI::addListener().

Object Lifetime

Some objects have their lifetime goverened by the user. For example, a **openni::VideoStream** is created using the **create()** method and destroyed by calling **destroy()**. In those cases, it is important that they get destroyed in the right order. For example, if you create a **openni::VideoStream** on a specific **openni::Device**, don't close the device before destroying that video stream. In the same manner, all **openni::VideoFrameRef** objects should be released before destroying the stream, and **all** OpenNI objects should be destroyed before calling **openni::OpenNI::shutdown()**.

Other objects are not created by the user, but are actually members of other objects. For example, calling **openni::Device::getDeviceInfo()** returns a reference to a **openni::DeviceInfo** object. Note that once the **openni::Device** object that was used gets closed, the **openni::DeviceInfo** reference is no longer valid. Do not try to use it to access methods of the **openni::DeviceInfo** object.

Arrays

Some methods return an array of items. We created a very simple template class, **openni::Array**, which only holds the pointer to the underlying C array and the number of elements in that array. Once the array has been released, the elements are no longer accessible. Do not keep a pointer to a specific item in the array and use it after releasing it.

Properties and Commands

OpenNI has a mechanism for generic properties and commands, both for devices and for streams. Commands are used by calling **openni::Device::invoke()** or **openni::VideoStream::invoke()** and Properties can be read via **openni::Device::getProperty()** and **openni::VideoStream::getProperty()** and written via **openni::Device::setProperty()** and **openni::VideoStream::setProperty()**.

Though both might use both primitives and complex types, generally, properties are used to encapsulate a logical data unit (even though complex) while commands are used in the following cases:

- The action requires some arguments (that are not part of the data unit).
- Obtaining a data unit via getProperty() have side effects.
- The operation is expensive enough that you want to communicate it to the user.
- The action has side effects.
- The data unit might be changed over time without an outside request.

Legal Stuff & Acknowledgments

The following open source components are included in OpenNI 2.X:

1. LibJPEG, 6b of 27-Mar-1998

Source code can be downloaded from: http://libjpeg.sourceforge.net/

This software is based in part on the work of the Independent JPEG Group. OpenNI 2.X source code also contains the source code of libjpeg, as well as its license terms. An online copy can be found at:

https://github.com/OpenNI/OpenNI2/blob/master/ThirdParty/LibJPEG/RE

This license can be found at:
https://github.com/aseprite/aseprite/blob/master/docs/licenses/libjpeg-

LICENSE. txt

2. LibUSB, 1.0.9, under GNU Lesser General Public License, v 2.1 (or any later version)

OpenNI 2.X uses a modified version of libusb, which was modified by PrimeSense on 2012.

The modified source code is distributed along with OpenNI 2.X source code and can be found in the following path: ThirdParty/PSCommon/XnLib/ThirdParty/libusb-1.0.9-Android

Full text of the license is available below.

3. GLUT 3.7.x

Some of the samples arrive with GLUT binaries and headers for Windows. Those are under the following copyright:

This license can be found at:
http://user.xmission.com/~nate/glut/README-win32.txt

the Lesser General Public License. In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system. Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library. The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run. GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you". A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables. The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".) "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library. Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does. 1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library. You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee. 2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions: a) The modified work must itself be a software library. b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change. c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License. d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that,in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.) These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library. In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License. 3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices. Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy. This option is useful when you wish to copy part of the code of the Library into a program that is not a library. 4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange. If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code. 5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License. However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables. When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law. If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.) Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself. 6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications. You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things: a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.) b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with. c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution. d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place. e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy. For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable. It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute. 7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things: a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above. b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work. 8. You may not copy, modify, sublicense, link with, or distribute the Library except as

Release Notes

OpenNI 2.2.0 Build 33 November 12 2013

Minimum Requirements: ---------------------

- Operating Systems:
  - Windows XP with SP2 and above, Windows 7, Windows 8, on x86 (32/64 bit)
  - Ubuntu 12.04 (32/64/arm) and above
  - Android 2.3 and above
  - Mac OSX 10.7 and above
- Processors:
  - Pentium 4, 1.4GHz and above
  - AMD Athlon 64/FX 1GHz and above
  - Arm Cortex A8 and above
- Memory: at least 64MB available.
- 250MB free hard disk space.
- Available USB 2.0 high-speed port.
- Development Environment:
  - Microsoft Visual Studio 2008 and 2010. The compiler can be MSVC compiler or an Intel Compiler 11 and above.
  - GCC 4.x
- Some of the sample applications require a graphics card equivalent to: ATI RADEON x1300 or NVIDIA GeForce 7300.

Notes: ------

- On Android, only native support (and samples) is currently provided. Please note that as bionic (Android linker) does not support the rpath option, the samples cannot start as is. To solve this, do one of the following:

- Copy OpenNI libraries (libOpenNI2.so, libPS1080.so and libOniFile.so) to /system/lib (requires root) - or -
- run `export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH` before starting the native executeable

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

openni::Array< T >	
openni::CameraSettings	
openni::CoordinateConverter	
openni::Device	
openni::OpenNI::DeviceConnectedListener	
openni::OpenNI::DeviceDisconnectedListener	
openni::DeviceInfo	
openni::OpenNI::DeviceStateChangedListener	
openni::VideoStream::FrameAllocator	
openni::VideoStream::NewFrameListener	
openni::OpenNI	
openni::PlaybackControl	
openni::Recorder	
openni::RGB888Pixel	
openni::SensorInfo	
openni::Version	
openni::VideoFrameRef	
openni::VideoMode	
openni::VideoStream	
openni::YUV422DoublePixel	

Public Member Functions

openni::Array< T > Class Template Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

	Array ()
	Array (const T *data, int count)
	~Array ()
int	**getSize** () const
const T &	**operator[]** (int index) const

Detailed Description

template<class T>
class openni::Array< T >

Provides a simple array class used throughout the API. Wraps a primitive array of objects, holding the elements and their count.

Constructor & Destructor Documentation

template<class T>

openni::Array< T >::Array ( ) `[inline]`

Default constructor. Creates an empty **Array** and sets the element count to zero.

template<class T>

openni::Array< T >::Array (const T * data, int count) `[inline]`

Constructor. Creates new **Array** from an existing primitive array of known size.

Template Parameters:

[in] T Object type this **Array** will contain.

Parameters:

`[in]`	**data**	Pointer to a primitive array of objects of type T.
`[in]`	**count**	Number of elements in the primitive array pointed to by data.

template<class T>

openni::Array< T >::~Array ( ) `[inline]`

Destructor. Destroys the **Array** object.

Member Function Documentation

template<class T>
int openni::Array< T >::getSize ( ) const `[inline]`

Getter function for the **Array** size.

Returns:
Current number of elements in the **Array**.

template<class T>
const T& openni::Array< T >::operator[] (int *index*) const `[inline]`

Implements the array indexing operator for the **Array** class.

The documentation for this class was generated from the following file:

- **OpenNI.h**

---

openni::CameraSettings Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

bool	**getAutoExposureEnabled** () const
bool	**getAutoWhiteBalanceEnabled** () const
int	**getExposure** ()
int	**getGain** ()
bool	**isValid** () const
Status	**setAutoExposureEnabled** (bool enabled)
Status	**setAutoWhiteBalanceEnabled** (bool enabled)
Status	**setExposure** (int exposure)
Status	**setGain** (int gain)

Friends

class	**VideoStream**

Member Function Documentation

bool openni::CameraSettings::getAutoExposureEnabled ( ) const

bool openni::CameraSettings::getAutoWhiteBalanceEnabled ( ) co

int openni::CameraSettings::getExposure ( ) `[inline]`

int openni::CameraSettings::getGain ( ) `[inline]`

bool openni::CameraSettings::isValid ( ) const `[inline]`

Status openni::CameraSettings::setAutoExposureEnabled (bool e

Status openni::CameraSettings::setAutoWhiteBalanceEnabled (bo

Status openni::CameraSettings::setExposure (int exposure) `[inli`

Status openni::CameraSettings::setGain (int gain) `[inline]`

Friends And Related Function Documentation

friend class VideoStream `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

openni::CoordinateConverter Class Reference

`#include <OpenNI.h>`

List of all members.

Static Public Member Functions

static **Status**	**convertDepthToColor** (const **VideoStream** &depthStream, const **VideoStream** &colorStream, int depthX, int depthY, **DepthPixel** depthZ, int *pColorX, int *pColorY)
static **Status**	**convertDepthToWorld** (const **VideoStream** &depthStream, int depthX, int depthY, **DepthPixel** depthZ, float *pWorldX, float *pWorldY, float *pWorldZ)
static **Status**	**convertDepthToWorld** (const **VideoStream** &depthStream, float depthX, float depthY, float depthZ, float *pWorldX, float *pWorldY, float *pWorldZ)
static **Status**	**convertWorldToDepth** (const **VideoStream** &depthStream, float worldX, float worldY, float worldZ, int *pDepthX, int *pDepthY, **DepthPixel** *pDepthZ)
static **Status**	**convertWorldToDepth** (const **VideoStream** &depthStream, float worldX, float worldY, float worldZ, float *pDepthX, float *pDepthY, float *pDepthZ)

Detailed Description

The **CoordinateConverter** class converts points between the different coordinate systems.

Depth and World coordinate systems

OpenNI applications commonly use two different coordinate systems to represent depth. These two systems are referred to as Depth and World representation.

Depth coordinates are the native data representation. In this system, the frame is a map (two dimensional array), and each pixel is assigned a depth value. This depth value represents the distance between the camera plane and whatever object is in the given pixel. The X and Y coordinates are simply the location in the map, where the origin is the top-left corner of the field of view.

World coordinates superimpose a more familiar 3D Cartesian coordinate system on the world, with the camera lens at the origin. In this system, every point is specified by 3 points -- x, y and z. The x axis of this system is along a line that passes through the infrared projector and CMOS imager of the camera. The y axis is parallel to the front face of the camera, and perpendicular to the x axis (it will also be perpendicular to the ground if the camera is upright and level). The z axis runs into the scene, perpendicular to both the x and y axis. From the perspective of the camera, an object moving from left to right is moving along the increasing x axis. An object moving up is moving along the increasing y axis, and an object moving away from the camera is moving along the increasing z axis.

Mathematically, the Depth coordinate system is the projection of the scene on the CMOS. If the sensor's angular field of view and resolution are known, then an angular size can be calculated for each pixel. This is how the conversion algorithms work. The dependence of this calculation on FoV and resolution is the reason that a **VideoStream** pointer must be provided to these functions. The **VideoStream** pointer is used to determine parameters for the specific points to be converted.

Since Depth coordinates are a projective, the apparent size of objects in depth coordinates (measured in pixels) will increase as an object moves closer to the sensor. The size of objects in the World coordinate system is independent of distance from the sensor.

Note that converting from Depth to World coordinates is relatively expensive computationally. It is generally not practical to convert the entire raw depth map to World coordinates. A better approach is to have your computer vision algorithm work in Depth coordinates for as long as possible, and only converting a few specific points to World coordinates right before output.

Note that when converting from Depth to World or vice versa, the Z value remains the same.

Member Function Documentation

static Status openni::CoordinateConverter::convertDepthToColor (

)

For a given depth point, provides the coordinates of the corresponding color value. Useful for superimposing the depth and color images. This operation is the same as turning on registration, but is performed on a single pixel rather than the whole image.

Parameters:

[in]	**depthStream**	Reference to a **openni::VideoStream** that produced the depth value
[in]	**colorStream**	Reference to a **openni::VideoStream** that we want to find the appropriate color pixel in
[in]	**depthX**	X value of the depth point, given in Depth coordinates and measured in pixels
[in]	**depthY**	Y value of the depth point, given in Depth coordinates and measured in pixels
[in]	**depthZ**	Z(depth) value of the depth point, given in the **PixelFormat** of depthStream
[out]	**pColorX**	The X coordinate of the color pixel that overlaps the given depth pixel, measured in pixels
[out]		

	pColorY	The Y coordinate of the color pixel that overlaps the given depth pixel, measured in pixels

static Status openni::CoordinateConverter::convertDepthToWorld (

)

Converts a single point from the Depth coordinate system to the World coordinate system.

Parameters:

[in]	**depthStream**	Reference to an openi::VideoStream that will be used to determine the format of the Depth coordinates
[in]	**depthX**	The X coordinate of the point to be converted, measured in pixels with 0 at the far left of the image
[in]	**depthY**	The Y coordinate of the point to be converted, measured in pixels with 0 at the top of the image
[in]	**depthZ**	the Z(depth) coordinate of the point to be converted, measured in the **PixelFormat** of depthStream
[out]	**pWorldX**	Pointer to a place to store the X coordinate of the output value, measured in millimeters in World coordinates
[out]	**pWorldY**	Pointer to a place to store the Y

		coordinate of the output value, measured in millimeters in World coordinates
[out]	**pWorldZ**	Pointer to a place to store the Z coordinate of the output value, measured in millimeters in World coordinates

static Status openni::CoordinateConverter::convertDepthToWorld (

)

Converts a single point from a floating point representation of the Depth coordinate system to the World coordinate system.

Parameters:

[in]	**depthStream**	Reference to an openi::VideoStream that will be used to determine the format of the Depth coordinates
[in]	**depthX**	The X coordinate of the point to be converted, measured in pixels with 0.0 at the far left of the image
[in]	**depthY**	The Y coordinate of the point to be converted, measured in pixels with 0.0 at the top of the image
[in]	**depthZ**	Z(depth) coordinate of the point to be converted, measured in the **PixelFormat** of depthStream
[out]	**pWorldX**	Pointer to a place to store the X

		coordinate of the output value, measured in millimeters in World coordinates
[out]	**pWorldY**	Pointer to a place to store the Y coordinate of the output value, measured in millimeters in World coordinates
[out]	**pWorldZ**	Pointer to a place to store the Z coordinate of the output value, measured in millimeters in World coordinates

static Status openni::CoordinateConverter::convertWorldToDepth (

)

Converts a single point from the World coordinate system to the Depth coordinate system.

Parameters:

[in]	**depthStream**	Reference to an **openni::VideoStream** that will be used to determine the format of the Depth coordinates
[in]	**worldX**	The X coordinate of the point to be converted, measured in millimeters in World coordinates
[in]	**worldY**	The Y coordinate of the point to be converted, measured in millimeters in World coordinates

[in]	**worldZ**	The Z coordinate of the point to be converted, measured in millimeters in World coordinates
[out]	**pDepthX**	Pointer to a place to store the X coordinate of the output value, measured in pixels with 0 at far left of image
[out]	**pDepthY**	Pointer to a place to store the Y coordinate of the output value, measured in pixels with 0 at top of image
[out]	**pDepthZ**	Pointer to a place to store the Z(depth) coordinate of the output value, measured in the **PixelFormat** of depthStream

static Status openni::CoordinateConverter::convertWorldToDepth (

)

Converts a single point from the World coordinate system to a floating point representation of the Depth coordinate system

Parameters:

[in]	**depthStream**	Reference to an **openni::VideoStream** that will be used to determine the format of the Depth coordinates
[in]	**worldX**	The X coordinate of the point to be converted, measured in millimeters in

		World coordinates
`[in]`	**worldY**	The Y coordinate of the point to be converted, measured in millimeters in World coordinates
`[in]`	**worldZ**	The Z coordinate of the point to be converted, measured in millimeters in World coordinates
`[out]`	**pDepthX**	Pointer to a place to store the X coordinate of the output value, measured in pixels with 0.0 at far left of the image
`[out]`	**pDepthY**	Pointer to a place to store the Y coordinate of the output value, measured in pixels with 0.0 at the top of the image
`[out]`	**pDepthZ**	Pointer to a place to store the Z(depth) coordinate of the output value, measured in millimeters with 0.0 at the camera lens

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Member Functions

openni::Device Class Reference

`#include <OpenNI.h>`

List of all members.

Public Member Functions

	Device ()
	Device (OniDeviceHandle handle)
	~Device ()
void	**close** ()
bool	**getDepthColorSyncEnabled** ()
const **DeviceInfo** &	**getDeviceInfo** () const
ImageRegistrationMode	**getImageRegistrationMode** () const
PlaybackControl *	**getPlaybackControl** ()
Status	**getProperty** (int propertyId, void *data, int *dataSize) const
template<class T > **Status**	**getProperty** (int propertyId, T *value) const
const **SensorInfo** *	**getSensorInfo** (**SensorType** sensorType)
bool	**hasSensor** (**SensorType** sensorType)
Status	**invoke** (int commandId, void *data, int dataSize)
template<class T > **Status**	**invoke** (int propertyId, T &value)
bool	**isCommandSupported** (int commandId) const
bool	**isFile** () const
bool	**isImageRegistrationModeSupported** (**ImageRegistrationMode** mode) const
bool	**isPropertySupported** (int propertyId) const
bool	**isValid** () const
Status	**open** (const char *uri)
Status	**setDepthColorSyncEnabled** (bool isEnabled)
Status	**setImageRegistrationMode** (**ImageRegistrationMode** mode)

Status	**setProperty** (int propertyId, const void *data, int dataSize)
template<class T >	
Status	**setProperty** (int propertyId, const T &value)

Detailed Description

The **Device** object abstracts a specific device; either a single hardware device, or a file device holding a recording from a hardware device. It offers the ability to connect to the device, and obtain information about its configuration and the data streams it can offer.

It provides the means to query and change all configuration parameters that apply to the device as a whole. This includes enabling depth/color image registration and frame synchronization.

Devices are used when creating and initializing **VideoStreams** -- you will need a valid pointer to a **Device** in order to use the **VideoStream.create()** function. This, along with configuration, is the primary use of this class for application developers.

Before devices can be created, **OpenNI::initialize()** must have been run to make the device drivers on the system available to the API.

Constructor & Destructor Documentation

openni::Device::Device ( ) `[inline]`

Default constructor. Creates a new empty **Device** object. This object will be invalid until it is initialized by calling its **open()** function.

openni::Device::Device (OniDeviceHandle handle) `[inline, explic`

Handle constructor. Creates a **Device** object based on the given initialized handle. This object will not destroy the underlying handle when **close()** or destructor is called

openni::Device::~Device ( ) `[inline]`

The destructor calls the **close()** function, but it is considered a best practice for applications to call **close()** manually on any **Device** that they run **open()** on.

Member Function Documentation

void openni::Device::close ( ) `[inline]`

Closes the device. This properly closes any files or shuts down hardware, as appropriate. This function is currently called by the destructor if not called manually by application code, but it is considered a best practice to manually close any device that was opened.

bool openni::Device::getDepthColorSyncEnabled ( ) `[inline]`

const DeviceInfo& openni::Device::getDeviceInfo ( ) const `[inline`

Provides information about this device in the form of a **DeviceInfo** object. This object can be used to access the URI of the device, as well as various USB descriptor strings that might be useful to an application.

Note that valid device info will not be available if this device has not yet been opened. If you are trying to obtain a URI to open a device, use **OpenNI::enumerateDevices()** instead.

Returns:

DeviceInfo object for this **Device**

ImageRegistrationMode openni::Device::getImageRegistrationMod

Gets the current image registration mode of this device. Image registration is used to properly superimpose two images from cameras located at different points in space. Please see the OpenNi 2.0 Programmer's Guide for more information about registration.

Returns:

Current image registration mode. See **ImageRegistrationMode** for possible return values.

PlaybackControl* openni::Device::getPlaybackControl ( ) `[inline]`

Gets an object through which playback of a file device can be controlled.

Returns:
NULL if this device is not a file device.

Status openni::Device::getProperty (int propertyId, void * data, int * dataSize) const `[inline]`

Get the value of a general property of the device. There are convenience functions for all the commonly used properties, such as image registration and frame synchronization. It is expected for this reason that this function will rarely be directly used by applications.

Parameters:

`[in]`	**propertyId**	Numerical ID of the property you would like to check.
`[out]`	**data**	Place to store the value of the property.
`[in,out]`	**dataSize**	IN: Size of the buffer passed in the `data` argument. OUT: the actual written size.

Returns:
Status code indicating results of this operation.

template<class T >

Status openni::Device::getProperty (int *propertyId*, T * *value*) const `[inline]`

Checks a property that provides an arbitrary data type as its output. It is not expected that application code will need this function frequently, as all commonly used properties have higher level functions provided.

Template Parameters:

[in] T Data type of the value to be read.

Parameters:

`[in]`	**propertyId**	The numerical ID of the property to be read.
`[in,out]`	**value**	Pointer to a place to store the value read from the property.

Returns:

Status code indicating success or failure of this operation.

const SensorInfo* openni::Device::getSensorInfo (SensorType sen

Get the SensorInfo for a specific sensor type on this device. The SensorInfo is useful primarily for determining which video modes are supported by the sensor.

Parameters:

`[in]`	**sensorType**	of sensor to get information about.

Returns:

SensorInfo object corresponding to the sensor type specified, or NULL if such a sensor is not available from this device.

bool openni::Device::hasSensor (SensorType *sensorType*) `[inlin`

This function checks to see if one of the specific sensor types defined in **SensorType** is available on this device. This allows an application to, for example, query for the presence of a depth sensor, or color sensor.

Parameters:

`[in]`	**sensorType**	of sensor to query for

Returns:
true if the **Device** supports the sensor queried, false otherwise.

Status openni::Device::invoke (int *commandId*, void * *data*, int *dataSize*) `[inline]`

Invokes a command that takes an arbitrary data type as its input. It is not expected that application code will need this function frequently, as all commonly used properties have higher level functions provided.

Parameters:

`[in]`	**commandId**	Numerical code of the property to be invoked.
`[in]`	**data**	Data to be passed to the property.
`[in]`	**dataSize**	size of the buffer passed in `data`.

Returns:
Status code indicating success or failure of this operation.

template<class T >
Status openni::Device::invoke (int *propertyId*, T & *value*) `[inline]`

Invokes a command that takes an arbitrary data type as its input. It is not expected that application code will need this function frequently, as all commonly used properties have higher level functions provided.

Template Parameters:

[in] T Type of data to be passed to the property.

Parameters:

`[in]`	**propertyId**	Numerical code of the property to be invoked.
`[in]`	**value**	Data to be passed to the property.

Returns:

Status code indicating success or failure of this operation.

bool openni::Device::isCommandSupported (int **commandId**) con

Checks if a specific command is supported by the device.

Parameters:

`[in]`	**commandId**	Command to be checked.

Returns:

true if the command is supported, false otherwise.

bool openni::Device::isFile ( ) const `[inline]`

Checks whether this device is a file device (i.e. a recording).

Returns:

true if this is a file device, false otherwise.

bool openni::Device::isImageRegistrationModeSupported (ImageR

Checks to see if this device can support registration of color video and depth video. Image registration is used to properly superimpose two images from cameras located at different points in space. Please see the OpenNi 2.0 Programmer's Guide for more information about registration.

Returns:
true if image registration is supported by this device, false otherwise.

bool openni::Device::isPropertySupported (int *propertyId*) const

Checks if a specific property is supported by the device.

Parameters:
`[in]` **propertyId** Property to be checked.

Returns:
true if the property is supported, false otherwise.

bool openni::Device::isValid ( ) const `[inline]`

Checks whether this **Device** object is currently connected to an actual file or hardware device.

Returns:
true if the **Device** is connected, false otherwise.

Status openni::Device::open (const char * *uri*) `[inline]`

Opens a device. This can either open a device chosen arbitrarily from all devices on the system, or open a specific device selected by passing this function the device URI.

To open any device, simply pass the constant**ANY_DEVICE** to this function. If multiple devices are connected to the system, then one of them will be opened. This procedure is most useful when it is known that exactly one device is (or can be) connected to the system. In that case, requesting a list of all devices and iterating through it would be a waste of effort.

If multiple devices are (or may be) connected to a system, then a URI will be required to select a specific device to open. There are two ways to obtain a URI: from a DeviceConnected event, or by calling **OpenNI::enumerateDevices()**.

In the case of a DeviceConnected event, the OpenNI::Listener will be provided with a **DeviceInfo** object as an argument to its onDeviceConnected() function. The **DeviceInfo.getUri()** function can then be used to obtain the URI.

If the application is not using event handlers, then it can also call the static function **OpenNI::enumerateDevices()**. This will return an array of **DeviceInfo** objects, one for each device currently available to the system. The application can then iterate through this list and select the desired device. The URI is again obtained via the **DeviceInfo::getUri()** function.

Standard codes of type Status are returned indicating whether opening was successful.

Parameters:

`[in]`	**uri**	String containing the URI of the device to be opened, or **ANY_DEVICE**.

Returns:

Status code with the outcome of the open operation.

Remarks:

For opening a recording file, pass the file path as a uri.

Status openni::Device::setDepthColorSyncEnabled (bool isEnable

Used to turn the depth/color frame synchronization feature on and off. When frame synchronization is enabled, the device will deliver depth and image frames that are separated in time by some maximum value. When disabled, the phase difference between depth and image frame generation cannot be guaranteed.

Parameters:

`[in]`	**isEnabled**	Set to TRUE to enable synchronization, FALSE to disable it

Returns:

Status code indicating success or failure of this operation

Status openni::Device::setImageRegistrationMode (ImageRegistra

Sets the image registration on this device. Image registration is used to properly superimpose two images from cameras located at different points in space. Please see the OpenNi 2.0 Programmer's Guide for more information about registration.

See **ImageRegistrationMode** for a list of valid settings to pass to this function.

It is a good practice to first check if the mode is supported by calling **isImageRegistrationModeSupported()**.

Parameters:

`[in]`	**mode**	Desired new value for the image registration mode.

Returns:

Status code for the operation.

Status openni::Device::setProperty (int *propertyId*, const void * *data*, int *dataSize*) `[inline]`

Sets the value of a general property of the device. There are convenience functions for all the commonly used properties, such as image registration and frame synchronization. It is expected for this reason that this function will rarely be directly used by applications.

Parameters:

`[in]`	**propertyId**	The numerical ID of the property to be set.
`[in]`	**data**	Place to store the data to be written to the property.
`[in]`	**dataSize**	Size of the data to be written to the property.

Returns:

Status code indicating results of this operation.

template<class T >
Status openni::Device::setProperty (int *propertyId*, const T & *value*) `[inline]`

Sets a property that takes an arbitrary data type as its input. It is not expected that application code will need this function frequently, as all commonly used properties have higher level functions provided.

Template Parameters:

T Type of data to be passed to the property.

Parameters:

`[in]`	**propertyId**	The numerical ID of the property to be set.
`[in]`	**value**	Place to store the data to be written to the

property.

Returns:
Status code indicating success or failure of this operation.

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Member Functions | Friends

openni::OpenNI::DeviceConnectedListener Class Reference

`#include <OpenNI.h>`

List of all members.

Public Member Functions

	DeviceConnectedListener ()
virtual	**~DeviceConnectedListener** ()
virtual void	**onDeviceConnected** (const **DeviceInfo** *)=0

Friends

class	**OpenNI**

Detailed Description

The **OpenNI::DeviceConnectedListener** class provides a means of registering for, and responding to when a device is connected.

onDeviceConnected is called whenever a new device is connected to the system (ie this event would be triggered when a new sensor is manually plugged into the host system running the application)

To use this class, you should write a new class that inherits from it, and override the onDeviceConnected method. Once you instantiate your class, use the **OpenNI::addDeviceConnectedListener()** function to add your listener object to OpenNI's list of listeners. Your handler function will then be called whenever the event occurs. A **OpenNI::removeDeviceConnectedListener()** function is also provided, if you want to have your class stop listening to these events for any reason.

Constructor & Destructor Documentation

openni::OpenNI::DeviceConnectedListener::DeviceConnectedListe

virtual openni::OpenNI::DeviceConnectedListener::~DeviceConnec

Member Function Documentation

virtual void openni::OpenNI::DeviceConnectedListener::onDeviceC

Callback function for the onDeviceConnected event. This function will be called whenever this event occurs. When this happens, a pointer to the **DeviceInfo** object for the newly connected device will be supplied. Note that once a device is removed, if it was opened by a **Device** object, that object can no longer be used to access the device, even if it was reconnected. Once a device was reconnected, **Device::open()** should be called again in order to use this device.

If you wish to open the new device as it is connected, simply query the provided **DeviceInfo** object to obtain the URI of the device, and pass this URI to the Device.Open() function.

Friends And Related Function Documentation

friend class OpenNI `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

openni::OpenNI::DeviceDisconnectedListener Class Reference

`#include <OpenNI.h>`

List of all members.

Public Member Functions

	DeviceDisconnectedListener ()
virtual	**~DeviceDisconnectedListener** ()
virtual void	**onDeviceDisconnected** (const **DeviceInfo** *)=0

Friends

class	**OpenNI**

Detailed Description

The **OpenNI::DeviceDisconnectedListener** class provides a means of registering for, and responding to when a device is disconnected.

onDeviceDisconnected is called when a device is removed from the system. Note that once a device is removed, if it was opened by a **Device** object, that object can no longer be used to access the device, even if it was reconnected. Once a device was reconnected, **Device::open()** should be called again in order to use this device.

To use this class, you should write a new class that inherits from it, and override the onDeviceDisconnected method. Once you instantiate your class, use the **OpenNI::addDeviceDisconnectedListener()** function to add your listener object to OpenNI's list of listeners. Your handler function will then be called whenever the event occurs. A **OpenNI::removeDeviceDisconnectedListener()** function is also provided, if you want to have your class stop listening to these events for any reason.

Constructor & Destructor Documentation

openni::OpenNI::DeviceDisconnectedListener::DeviceDisconnecte

virtual openni::OpenNI::DeviceDisconnectedListener::~DeviceDisc

Member Function Documentation

virtual void openni::OpenNI::DeviceDisconnectedListener::onDevi

Callback function for the onDeviceDisconnected event. This function will be called whenever this event occurs. When this happens, a pointer to the **DeviceInfo** object for the newly disconnected device will be supplied. Note that once a device is removed, if it was opened by a **Device** object, that object can no longer be used to access the device, even if it was reconnected. Once a device was reconnected, **Device::open()** should be called again in order to use this device.

Friends And Related Function Documentation

friend class OpenNI `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Member Functions | Friends

openni::DeviceInfo Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

const char *	**getName** () const
const char *	**getUri** () const
uint16_t	**getUsbProductId** () const
uint16_t	**getUsbVendorId** () const
const char *	**getVendor** () const

Friends

class	**Device**
class	**OpenNI**

Detailed Description

The **DeviceInfo** class encapsulates info related to a specific device.

Applications will generally obtain objects of this type via calls to **OpenNI::enumerateDevices()** or **openni::Device::getDeviceInfo()**, and then use the various accessor functions to obtain specific information on that device.

There should be no reason for application code to instantiate this object directly.

Member Function Documentation

const char* openni::DeviceInfo::getName ( ) const `[inline]`

Returns the device name for this device.

const char* openni::DeviceInfo::getUri ( ) const `[inline]`

Returns the device URI. URI can be used by **Device::open** to open a specific device. The URI string format is determined by the driver.

uint16_t openni::DeviceInfo::getUsbProductId ( ) const `[inline]`

Returns the USB PID code for this device.

uint16_t openni::DeviceInfo::getUsbVendorId ( ) const `[inline]`

Returns the USB VID code for this device.

const char* openni::DeviceInfo::getVendor ( ) const `[inline]`

Returns a the vendor name for this device.

Friends And Related Function Documentation

friend class Device `[friend]`

friend class OpenNI `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

openni::OpenNI::DeviceStateChangedListener Class Reference

`#include <OpenNI.h>`

List of all members.

Public Member Functions

	DeviceStateChangedListener ()
virtual	**~DeviceStateChangedListener** ()
virtual void	**onDeviceStateChanged** (const **DeviceInfo** *, **DeviceState**)=0

Friends

class	**OpenNI**

Detailed Description

The **OpenNI::DeviceStateChangedListener** class provides a means of registering for, and responding to when a device's state is changed.

onDeviceStateChanged is triggered whenever the state of a connected device is changed.

To use this class, you should write a new class that inherits from it, and override the onDeviceStateChanged method. Once you instantiate your class, use the **OpenNI::addDeviceStateChangedListener()** function to add your listener object to OpenNI's list of listeners. Your handler function will then be called whenever the event occurs. A **OpenNI::removeDeviceStateChangedListener()** function is also provided, if you want to have your class stop listening to these events for any reason.

Constructor & Destructor Documentation

openni::OpenNI::DeviceStateChangedListener::DeviceStateChang

virtual openni::OpenNI::DeviceStateChangedListener::~DeviceStat

Member Function Documentation

virtual void openni::OpenNI::DeviceStateChangedListener::onDevi

Callback function for the onDeviceStateChanged event. This function will be called whenever this event occurs. When this happens, a pointer to a **DeviceInfo** object for the affected device will be supplied, as well as the new DeviceState value of that device.

Friends And Related Function Documentation

friend class OpenNI `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

---

openni::VideoStream::FrameAllocator Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

virtual	**~FrameAllocator** ()
virtual void *	**allocateFrameBuffer** (int size)=0
virtual void	**freeFrameBuffer** (void *data)=0

Friends

class	**VideoStream**

Constructor & Destructor Documentation

virtual openni::VideoStream::FrameAllocator::~FrameAllocator ( )

Member Function Documentation

virtual void* openni::VideoStream::FrameAllocator::allocateFrameI

virtual void openni::VideoStream::FrameAllocator::freeFrameBuffe

Friends And Related Function Documentation

friend class VideoStream `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

---

openni::VideoStream::NewFrameListener Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

	NewFrameListener ()
virtual	**~NewFrameListener** ()
virtual void	**onNewFrame** (**VideoStream** &)=0

Friends

class	**VideoStream**

Detailed Description

The **VideoStream::NewFrameListener** class is provided to allow the implementation of event driven frame reading. To use it, create a class that inherits from it and implement override the **onNewFrame()** method. Then, register your created class with an active **VideoStream** using the **VideoStream::addNewFrameListener()** function. Once this is done, the event handler function you implemented will be called whenever a new frame becomes available. You may call **VideoStream::readFrame()** from within the event handler.

Constructor & Destructor Documentation

openni::VideoStream::NewFrameListener::NewFrameListener ( ) `[i`

Default constructor.

virtual openni::VideoStream::NewFrameListener::~NewFrameListe

Member Function Documentation

virtual void openni::VideoStream::NewFrameListener::onNewFram

Derived classes should implement this function to handle new frames.

Friends And Related Function Documentation

friend class VideoStream `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

Classes |
Static Public Member Functions

openni::OpenNI Class Reference

`#include <OpenNI.h>`

List of all members.

Classes

class	**DeviceConnectedListener**
class	**DeviceDisconnectedListener**
class	**DeviceStateChangedListener**

Static Public Member Functions

static **Status**	**addDeviceConnectedListener** (**DeviceConnectedListener** *pListener)
static **Status**	**addDeviceDisconnectedListener** (**DeviceDisconnectedListener** *pListener)
static **Status**	**addDeviceStateChangedListener** (**DeviceStateChangedListener** *pListener)
static void	**enumerateDevices** (**Array**< **DeviceInfo** > *deviceInfoList)
static const char *	**getExtendedError** ()
static **Status**	**getLogFileName** (char *strFileName, int nBufferSize)
static **Version**	**getVersion** ()
static **Status**	**initialize** ()
static void	**removeDeviceConnectedListener** (**DeviceConnectedListener** *pListener)
static void	**removeDeviceDisconnectedListener** (**DeviceDisconnectedListener** *pListener)
static void	**removeDeviceStateChangedListener** (**DeviceStateChangedListener** *pListener)
static **Status**	**setLogAndroidOutput** (bool bAndroidOutput)
static **Status**	**setLogConsoleOutput** (bool bConsoleOutput)
static **Status**	**setLogFileOutput** (bool bFileOutput)
static **Status**	**setLogMinSeverity** (int nMinSeverity)
static **Status**	**setLogOutputFolder** (const char *strLogOutputFolder)
static void	**shutdown** ()
static **Status**	**waitForAnyStream** (**VideoStream** **pStreams, int streamCount, int *pReadyStreamIndex, int timeout=**TIMEOUT_FOREVER**)

Detailed Description

The **OpenNI** class is a static entry point to the library. It is used by every **OpenNI** 2.0 application to initialize the SDK and drivers to enable creation of valid device objects.

It also defines a listener class and events that enable for event driven notification of device connection, device disconnection, and device configuration changes.

In addition, it gives access to SDK version information and provides a function that allows you to wait for data to become available on any one of a list of streams (as opposed to waiting for data on one specific stream with functions provided by the **VideoStream** class)

Member Function Documentation

static Status openni::OpenNI::addDeviceConnectedListener (Devic

Add a listener to the list of objects that receive the event when a device is connected. See the **OpenNI::DeviceConnectedListener** class for details on utilizing the events provided by **OpenNI**.

Parameters:

pListener	Pointer to the Listener to be added to the list

Returns:

Status code indicating success or failure of this operation.

static Status openni::OpenNI::addDeviceDisconnectedListener (De

Add a listener to the list of objects that receive the event when a device is disconnected. See the **OpenNI::DeviceDisconnectedListener** class for details on utilizing the events provided by **OpenNI**.

Parameters:

pListener	Pointer to the Listener to be added to the list

Returns:

Status code indicating success or failure of this operation.

static Status openni::OpenNI::addDeviceStateChangedListener (De

Add a listener to the list of objects that receive the event when a device's state changes. See the **OpenNI::DeviceStateChangedListener** class for details on utilizing the events provided by **OpenNI**.

Parameters:

pListener Pointer to the Listener to be added to the list

Returns:

Status code indicating success or failure of this operation.

static void openni::OpenNI::enumerateDevices (Array< DeviceInfo

Fills up an array of **DeviceInfo** objects with devices that are available.

Parameters:

`[in,out]` **deviceInfoList** An array to be filled with devices.

static const char* openni::OpenNI::getExtendedError ( ) `[inline, s`

Retrieves the calling thread's last extended error information. The last extended error information is maintained on a per-thread basis. Multiple threads do not overwrite each other's last extended error information.

The extended error information is cleared on every call to an **OpenNI** method, so you should call this method immediately after a call to an **OpenNI** method which have failed.

static Status openni::OpenNI::getLogFileName (char * strFileNam
int nBufferSize
) `[inline, st`

Get current log file name

Parameters:

char * strFileName [out] returned file name buffer

int	nBufferSize [in] Buffer size

Return values:

STATUS_OK	Upon successful completion.
STATUS_ERROR	Upon any kind of failure.

static Version openni::OpenNI::getVersion ( ) `[inline, static]`

Returns the version of **OpenNI**

static Status openni::OpenNI::initialize ( ) `[inline, static]`

Initialize the library. This will load all available drivers, and see which devices are available It is forbidden to call any other method in **OpenNI** before calling **initialize()**.

static void openni::OpenNI::removeDeviceConnectedListener (Dev

Remove a listener from the list of objects that receive the event when a device is connected. See the **OpenNI::DeviceConnectedListener** class for details on utilizing the events provided by **OpenNI**.

Parameters:

pListener	Pointer to the Listener to be removed from the list

Returns:

Status code indicating the success or failure of this operation.

static void openni::OpenNI::removeDeviceDisconnectedListener (I

Remove a listener from the list of objects that receive the event when a device is disconnected. See the

OpenNI::DeviceDisconnectedListener class for details on utilizing the events provided by OpenNI.

Parameters:

pListener	Pointer to the Listener to be removed from the list

Returns:

Status code indicating the success or failure of this operation.

static void openni::OpenNI::removeDeviceStateChangedListener (

Remove a listener from the list of objects that receive the event when a device's state changes. See the OpenNI::DeviceStateChangedListener class for details on utilizing the events provided by OpenNI.

Parameters:

pListener	Pointer to the Listener to be removed from the list

Returns:

Status code indicating the success or failure of this operation.

static Status openni::OpenNI::setLogAndroidOutput (bool bAndro

Configures if log entries will be printed to the Android log.

Parameters:

OniBool	bAndroidOutput bAndroidOutput [in] TRUE to print log entries to the Android log, FALSE otherwise.

Return values:

STATUS_OK	Upon successful completion.
STATUS_ERROR	Upon any kind of failure.

static Status openni::OpenNI::setLogConsoleOutput (bool bConso

Configures if log entries will be printed to console.

Parameters:

const	OniBool bConsoleOutput [in] TRUE to print log entries to console, FALSE otherwise.

Return values:

STATUS_OK	Upon successful completion.
STATUS_ERROR	Upon any kind of failure.

static Status openni::OpenNI::setLogFileOutput (bool bFileOutput

Configures if log entries will be printed to file.

Parameters:

const	OniBool bConsoleOutput [in] TRUE to print log entries to file, FALSE otherwise.

Return values:

STATUS_OK	Upon successful completion.
STATUS_ERROR	Upon any kind of failure.

static Status openni::OpenNI::setLogMinSeverity (int nMinSeverity

Set minimum severity for log produce

Parameters:

const	char * strMask [in] Logger name
int	nMinSeverity [in] Logger severity

Return values:

STATUS_OK	Upon successful completion.

STATUS_ERROR	Upon any kind of failure.

static Status openni::OpenNI::setLogOutputFolder (const char * st

Change the log output folder

Parameters:

const	char * strLogOutputFolder [in] log required folder

Return values:

STATUS_OK	Upon successful completion.
STATUS_ERROR	Upon any kind of failure.

static void openni::OpenNI::shutdown ( ) `[inline, static]`

Stop using the library. Unload all drivers, close all streams and devices. Once shutdown was called, no other calls to OpenNI is allowed.

static Status openni::OpenNI::waitForAnyStream (VideoStream ** int int * int)

Wait for a new frame from any of the streams provided. The function blocks until any of the streams has a new frame available, or the timeout has passed.

Parameters:

`[in]`	pStreams	An array of streams to wait for.
`[in]`	streamCount	The number of streams in `pStreams`

`[out]`	**pReadyStreamIndex**	The index of the first stream that has new frame available.
`[in]`	**timeout**	[Optional] A timeout before returning if no stream has new data. Default value is **TIMEOUT_FOREVER**.

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Member Functions | Friends

openni::PlaybackControl Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

	~PlaybackControl ()
int	**getNumberOfFrames** (const **VideoStream** &stream) const
bool	**getRepeatEnabled** () const
float	**getSpeed** () const
bool	**isValid** () const
Status	**seek** (const **VideoStream** &stream, int frameIndex)
Status	**setRepeatEnabled** (bool repeat)
Status	**setSpeed** (float speed)

Friends

class	**Device**

Detailed Description

The **PlaybackControl** class provides access to a series of specific to playing back a recording from a file device.

When playing a stream back from a recording instead of playing from a live device, it is possible to vary playback speed, change the current time location (ie fast forward / rewind / seek), specify whether the playback should be repeated at the end of the recording, and query the total size of the recording.

Since none of these functions make sense in the context of a physical device, they are split out into a seperate playback control class. To use, simply create your file device, create a **PlaybackControl**, and then attach the **PlaybackControl** to the file device.

Constructor & Destructor Documentation

openni::PlaybackControl::~PlaybackControl ( ) `[inline]`

Deconstructor. Destroys a **PlaybackControl** class. The deconstructor presently detaches from its recording automatically, but it is considered a best practice for applications to manually detach from any stream that was attached to.

Member Function Documentation

int openni::PlaybackControl::getNumberOfFrames (const VideoStr

Provides the a count of frames that this recording contains for a given stream. This is useful both to determine the length of the recording, and to ensure that a valid Frame Index is set when using the **PlaybackControl::seek()** function.

Parameters:

`[in]` **stream** The video stream to count frames for

Returns:

Number of frames in provided **VideoStream**, or 0 if the stream is not part of the recording

bool openni::PlaybackControl::getRepeatEnabled ( ) const `[inline`

Gets the current repeat setting of the file device.

Returns:

true if repeat is enabled, false if not enabled.

float openni::PlaybackControl::getSpeed ( ) const `[inline]`

Getter function for the current playback speed of this device.

This value is expressed as a multiple of the speed the original recording was taken at. For example, if the original recording was at 30fps, and playback speed is set to 0.5, then the recording will play at 15fps. If playback speed is set to 2.0, then the recording would playback at 60fps.

In addition, there are two "special" values. A playback speed of 0.0

indicates that the playback should occur as fast as the system is capable of returning frames. This is most useful when testing algorithms on large datasets, as it enables playback to be done at a much higher rate than would otherwise be possible.

A value of -1 indicates that speed is "manual". In this mode, new frames will only become available when an application manually reads them. If used in a polling loop, this setting also enables systems to read and process frames limited only by available processing speeds.

Returns:
Current playback speed of the device, measured as ratio of recording speed.

bool openni::PlaybackControl::isValid ( ) const `[inline]`

Status openni::PlaybackControl::seek (const VideoStream & strea
int fram
) `[inli`

Seeks within a **VideoStream** to a given FrameID. Note that when this function is called on one stream, all other streams will also be changed to the corresponding place in the recording. The FrameIDs of different streams may not match, since FrameIDs may differ for streams that are not synchronized, but the recording will set all streams to the same moment in time.

Parameters:

`[in]`	**stream**	Stream for which the frameIndex value is valid.
`[in]`	**frameIndex**	Frame index to move playback to

Returns:
Status code indicating success or failure of this operation

Status openni::PlaybackControl::setRepeatEnabled (bool *repeat*)

Changes the current repeat mode of the device. If repeat mode is turned on, then the recording will begin playback again at the beginning after the last frame is read. If turned off, no more frames will become available after last frame is read.

Parameters:

`[in]`	**repeat**	New value for repeat -- true to enable, false to disable

Returns:

Status code indicating success or failure of this operations.

Status openni::PlaybackControl::setSpeed (float *speed*) `[inline]`

Setter function for the playback speed of the device. For a full explaination of what this value means

See also:

PlaybackControl::getSpeed().

Parameters:

`[in]`	**speed**	Desired new value of playback speed, as ratio of original recording.

Returns:

Status code indicating success or failure of this operation.

Friends And Related Function Documentation

friend class Device `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Member Functions

openni::Recorder Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

	Recorder ()
	~Recorder ()
Status	**attach** (**VideoStream** &stream, bool allowLossyCompression=false)
Status	**create** (const char *fileName)
void	**destroy** ()
bool	**isValid** () const
Status	**start** ()
void	**stop** ()

Detailed Description

The **Recorder** class is used to record streams to an ONI file.

After a recorder is instantiated, it must be initialized with a specific filename where the recording will be stored. The recorder is then attached to one or more streams. Once this is complete, the recorder can be told to start recording. The recorder will store every frame from every stream to the specified file. Later, this file can be used to initialize a file **Device**, and used to play back the same data that was recorded.

Opening a file device is done by passing its path as the uri to the **Device::open()** method.

See also:
PlaybackControl for options available to play a reorded file.

Constructor & Destructor Documentation

openni::Recorder::Recorder ( ) `[inline]`

Creates a recorder. The recorder is not valid, i.e. **isValid()** returns false. You must initialize the recorder before use with **create()**.

openni::Recorder::~Recorder ( ) `[inline]`

Destroys a recorder. This will also stop recording.

Member Function Documentation

Status openni::Recorder::attach (VideoStream & stream,
bool allowLossyComp
) `[inline]`

Attaches a stream to the recorder. Note, this won't start recording, you should explicitly start it using **start()** method. As soon as the recording process has been started, no more streams can be attached to the recorder.

Parameters:

`[in]`	**stream**	The stream to be recorded.
`[in]`	**allowLossyCompression**	[Optional] If this value is true, the recorder might use a lossy compression, which means that when the recording will be played-back, there might be small differences from the original frame. Default value is false.

Status openni::Recorder::create (const char * fileName) `[inline]`

Initializes a recorder. You can initialize the recorder only once. Attempts to intialize more than once will result in an error code being returned.

Initialization assigns the recorder to an output file that will be used for recording. Before use, the **attach()** function must also be used to assign input data to the **Recorder**.

Parameters:

`[in]`	**fileName**	The name of a file which will contain the

recording.

Returns:
Status code which indicates success or failure of the operation.

void openni::Recorder::destroy ( ) `[inline]`

Destroys the recorder object.

bool openni::Recorder::isValid ( ) const `[inline]`

Verifies if the recorder is valid, i.e. if one can record with this recorder. A recorder object is not valid until the **create()** method is called.

Returns:
true if the recorder has been intialized, false otherwise.

Status openni::Recorder::start ( ) `[inline]`

Starts recording. Once this method is called, the recorder will take all subsequent frames from the attached streams and store them in the file. You may not attach additional streams once recording was started.

void openni::Recorder::stop ( ) `[inline]`

Stops recording. You may use **start()** to resume the recording.

The documentation for this class was generated from the following file:

- **OpenNI.h**

openni::RGB888Pixel Struct Reference

```
#include <OpenNI.h>
```

List of all members.

Public Attributes

uint8_t	**b**
uint8_t	**g**
uint8_t	**r**

Detailed Description

Holds the value of a single color image pixel in 24-bit RGB format.

Member Data Documentation

uint8_t openni::RGB888Pixel::b

uint8_t openni::RGB888Pixel::g

uint8_t openni::RGB888Pixel::r

The documentation for this struct was generated from the following file:

- **OpenNI.h**

Public Member Functions | Friends

openni::SensorInfo Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

SensorType	**getSensorType** () const
const **Array**< **VideoMode** > &	**getSupportedVideoModes** () const

Friends

class	**Device**
class	**VideoStream**

Detailed Description

The **SensorInfo** class encapsulates all info related to a specific sensor in a specific device. A **Device** object holds a **SensorInfo** object for each sensor it contains. A **VideoStream** object holds one **SensorInfo** object, describing the sensor used to produce that stream.

A given **SensorInfo** object will contain the type of the sensor (Depth, IR or Color), and a list of all video modes that the sensor can support. Each available video mode will have a single **VideoMode** object that can be queried to get the details of that mode.

SensorInfo objects should be the only source of **VideoMode** objects for the vast majority of application programs.

Application programs will never directly instantiate objects of type **SensorInfo**. In fact, no public constructors are provided. **SensorInfo** objects should be obtained either from a **Device** or **VideoStream**, and in turn be used to provide available video modes for that sensor.

Member Function Documentation

SensorType openni::SensorInfo::getSensorType ( ) const `[inline]`

Provides the sensor type of the sensor this object is associated with.

Returns:
Type of the sensor.

const Array<VideoMode>& openni::SensorInfo::getSupportedVide

Provides a list of video modes that this sensor can support. This function is the recommended method to be used by applications to obtain **VideoMode** objects.

Returns:
Reference to an array of **VideoMode** objects, one for each supported video mode.

Friends And Related Function Documentation

friend class Device `[friend]`

friend class VideoStream `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Attributes

openni::Version Struct Reference

```
#include <OpenNI.h>
```

List of all members.

Public Attributes

int	**build**
int	**maintenance**
int	**major**
int	**minor**

Detailed Description

Holds an **OpenNI** version number, which consists of four separate numbers in the format: `major.minor.maintenance.build`. For example: 2.0.0.20.

Member Data Documentation

int openni::Version::build

Build number. Incremented for each new API build. Generally not shown on the installer and download site.

int openni::Version::maintenance

Maintenance build number, incremented for new releases that primarily provide minor bug fixes.

int openni::Version::major

Major version number, incremented for major API restructuring.

int openni::Version::minor

Minor version number, incremented when significant new features added.

The documentation for this struct was generated from the following file:

- **OpenNI.h**

openni::VideoFrameRef Class Reference

`#include <OpenNI.h>`

List of all members.

Public Member Functions

	VideoFrameRef ()
	VideoFrameRef (const **VideoFrameRef** &other)
	~VideoFrameRef ()
int	**getCropOriginX** () const
int	**getCropOriginY** () const
bool	**getCroppingEnabled** () const
const void *	**getData** () const
int	**getDataSize** () const
int	**getFrameIndex** () const
int	**getHeight** () const
SensorType	**getSensorType** () const
int	**getStrideInBytes** () const
uint64_t	**getTimestamp** () const
const **VideoMode** &	**getVideoMode** () const
int	**getWidth** () const
bool	**isValid** () const
VideoFrameRef &	**operator=** (const **VideoFrameRef** &other)
void	**release** ()

Friends

class	**VideoStream**

Detailed Description

The **VideoFrameRef** class encapsulates a single video frame - the output of a **VideoStream** at a specific time. The data contained will be a single frame of color, IR, or depth video, along with associated meta data.

An object of type **VideoFrameRef** does not actually hold the data of the frame, but only a reference to it. The reference can be released by destroying the **VideoFrameRef** object, or by calling the **release()** method. The actual data of the frame is freed when the last reference to it is released.

The usual way to obtain **VideoFrameRef** objects is by a call to **VideoStream**.:readFrame().

All data references by a **VideoFrameRef** is stored as a primitive array of pixels. Each pixel will be of a type according to the configured pixel format (see **VideoMode**).

Constructor & Destructor Documentation

openni::VideoFrameRef::VideoFrameRef ( ) `[inline]`

Default constructor. Creates a new empty **VideoFrameRef** object. This object will be invalid until initialized by a call to **VideoStream::readFrame()**.

openni::VideoFrameRef::~VideoFrameRef ( ) `[inline]`

Destroy this object and release the reference to the frame.

openni::VideoFrameRef::VideoFrameRef (const VideoFrameRef &

Copy constructor. Creates a new **VideoFrameRef** object. The newly created object will reference the same frame current object references.

Parameters:

`[in]` **other** Another **VideoFrameRef** object.

Member Function Documentation

int openni::VideoFrameRef::getCropOriginX ( ) const `[inline]`

Indicates the X coordinate of the upper left corner of the crop window.

Returns:
Distance of crop origin from left side of image, in pixels.

int openni::VideoFrameRef::getCropOriginY ( ) const `[inline]`

Indicates the Y coordinate of the upper left corner of the crop window.

Returns:
Distance of crop origin from top of image, in pixels.

bool openni::VideoFrameRef::getCroppingEnabled ( ) const `[inlin`

Indicates whether cropping was enabled when the frame was produced.

Returns:
true if cropping is enabled, false otherwise

const void* openni::VideoFrameRef::getData ( ) const `[inline]`

Getter function for the array of data pointed to by this object.

Returns:
Pointer to the actual frame data array. Type of data pointed to can be determined according to the pixel format (can be obtained by calling **getVideoMode()**).

int openni::VideoFrameRef::getDataSize ( ) const `[inline]`

Getter function for the size of the data contained by this object. Useful primarily when allocating buffers.

Returns:
Current size of data pointed to by this object, measured in bytes.

int openni::VideoFrameRef::getFrameIndex ( ) const `[inline]`

Frames are provided sequential frame ID numbers by the sensor that produced them. If frame synchronization has been enabled for a device via **Device::setDepthColorSyncEnabled()**, then frame numbers for corresponding frames of depth and color are guaranteed to match.

If frame synchronization is not enabled, then there is no guarantee of matching frame indexes between **VideoStreams**. In the latter case, applications should use timestamps instead of frame indexes to align frames in time.

Returns:
Index number for this frame.

int openni::VideoFrameRef::getHeight ( ) const `[inline]`

Gives the current height of this frame, measured in pixels. If cropping is enabled, this will be the length of the cropping window. If cropping is not enabled, then this will simply be equal to the Y resolution of the **VideoMode** used to produce this frame.

SensorType openni::VideoFrameRef::getSensorType ( ) const `[inl`

Getter function for the sensor type used to produce this frame. Used to determine whether this is an IR, Color or Depth frame. See the **SensorType** enumeration for all possible return values from this function.

Returns:
The type of sensor used to produce this frame.

int openni::VideoFrameRef::getStrideInBytes ( ) const `[inline]`

Gives the length of one row of pixels, measured in bytes. Primarily useful for indexing the array which contains the data.

Returns:
Stride of the array which contains the image for this frame, in bytes

uint64_t openni::VideoFrameRef::getTimestamp ( ) const `[inline]`

Provides a timestamp for the frame. The 'zero' point for this stamp is implementation specific, but all streams from the same device are guaranteed to use the same zero. This value can therefore be used to compute time deltas between frames from the same device, regardless of whether they are from the same stream.

Returns:
Timestamp of frame, measured in microseconds from an arbitrary zero

const VideoMode& openni::VideoFrameRef::getVideoMode ( ) cons

Returns a reference to the **VideoMode** object assigned to this frame. This object describes the video mode the sensor was configured to when the frame was produced and can be used to determine the pixel format and resolution of the data. It will also provide the frame rate that the sensor was running at when it recorded this frame.

Returns:
Reference to the VideoMode assigned to this frame.

int openni::VideoFrameRef::getWidth ( ) const `[inline]`

Gives the current width of this frame, measured in pixels. If cropping is enabled, this will be the width of the cropping window. If cropping is not enabled, then this will simply be equal to the X resolution of the VideoMode used to produce this frame.

Returns:
Width of this frame in pixels.

bool openni::VideoFrameRef::isValid ( ) const `[inline]`

Check if this object references an actual frame.

VideoFrameRef& openni::VideoFrameRef::operator= (const VideoF

Make this VideoFrameRef object reference the same frame that the `other` frame references. If this object referenced another frame before calling this method, the previous frame will be released.

Parameters:
`[in]` **other** Another VideoFrameRef object.

void openni::VideoFrameRef::release ( ) `[inline]`

Release the reference to the frame. Once this method is called, the object becomes invalid, and no method should be called other than the assignment operator, or passing this object to a **VideoStream::readFrame()** call.

Friends And Related Function Documentation

friend class VideoStream `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

Public Member Functions | Friends

openni::VideoMode Class Reference

```
#include <OpenNI.h>
```

List of all members.

Public Member Functions

	VideoMode ()
	VideoMode (const **VideoMode** &other)
int	**getFps** () const
PixelFormat	**getPixelFormat** () const
int	**getResolutionX** () const
int	**getResolutionY** () const
VideoMode &	**operator=** (const **VideoMode** &other)
void	**setFps** (int fps)
void	**setPixelFormat** (**PixelFormat** format)
void	**setResolution** (int resolutionX, int resolutionY)

Friends

class	**SensorInfo**
class	**VideoFrameRef**
class	**VideoStream**

Detailed Description

Encapsulates a group of settings for a **VideoStream**. Settings stored include frame rate, resolution, and pixel format.

This class is used as an input for changing the settings of a **VideoStream**, as well as an output for reporting the current settings of that class. It is also used by **SensorInfo** to report available video modes of a stream.

Recommended practice is to use **SensorInfo::getSupportedVideoModes()** to obtain a list of valid video modes, and then to use items from that list to pass new settings to **VideoStream**. This is much less likely to produce an invalid video mode than instantiating and manually changing objects of this class.

Constructor & Destructor Documentation

openni::VideoMode::VideoMode ( ) `[inline]`

Default constructor, creates an empty **VideoMode** object. Application programs should, in most cases, use the copy constructor to copy an existing valid video mode. This is much less error prone that creating and attempting to configure a new **VideoMode** from scratch.

openni::VideoMode::VideoMode (const VideoMode & other) `[inli`

Copy constructor, creates a new **VideoMode** identical to an existing **VideoMode**.

Parameters:

`[in]` **other** Existing **VideoMode** to copy.

Member Function Documentation

int openni::VideoMode::getFps ( ) const `[inline]`

Getter function for the frame rate of this VideoMode.

Returns:
Current frame rate, measured in frames per second.

PixelFormat openni::VideoMode::getPixelFormat ( ) const `[inline]`

Getter function for the pixel format of this VideoMode.

Returns:
Current pixel format setting of this VideoMode.

int openni::VideoMode::getResolutionX ( ) const `[inline]`

Getter function for the X resolution of this VideoMode.

Returns:
Current horizontal resolution of this VideoMode, in pixels.

int openni::VideoMode::getResolutionY ( ) const `[inline]`

Getter function for the Y resolution of this VideoMode.

Returns:
Current vertical resolution of this VideoMode, in pixels.

VideoMode& openni::VideoMode::operator= (const VideoMode &

Assignment operator. Sets the pixel format, frame rate, and resolution of this **VideoMode** to equal that of a different **VideoMode**.

Parameters:

`[in]` **other** Existing **VideoMode** to copy settings from.

void openni::VideoMode::setFps (int *fps*) `[inline]`

Setter function for the frame rate. Application use of this function is not recommended. Instead, use **SensorInfo::getSupportedVideoModes()** to obtain a list of valid video modes.

Parameters:

`[in]` **fps** Desired new frame rate, measured in frames per second.

void openni::VideoMode::setPixelFormat (PixelFormat *format*) `[in`

Setter function for the pixel format of this **VideoMode**. Application use of this function is not recommended. Instead, use **SensorInfo::getSupportedVideoModes()** to obtain a list of valid video modes.

Parameters:

`[in]` **format** Desired new pixel format for this **VideoMode**.

void openni::VideoMode::setResolution (int *resolutionX*, int *resolutionY*) `[inline]`

Setter function for the resolution of this **VideoMode**. Application use of this function is not recommended. Instead, use **SensorInfo::getSupportedVideoModes()** to obtain a list of valid video modes.

Parameters:

`[in]`	**resolutionX**	Desired new horizontal resolution in pixels.
`[in]`	**resolutionY**	Desired new vertical resolution in pixels.

Friends And Related Function Documentation

friend class SensorInfo `[friend]`

friend class VideoFrameRef `[friend]`

friend class VideoStream `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

Classes | Public Member Functions | Friends

openni::VideoStream Class Reference

```
#include <OpenNI.h>
```

List of all members.

Classes

class	**FrameAllocator**
class	**NewFrameListener**

Public Member Functions

	VideoStream ()
	VideoStream (OniStreamHandle handle)
	~VideoStream ()
Status	**addNewFrameListener** (**NewFrameListener** *pListener)
Status	**create** (const **Device** &device, **SensorType** sensorType)
void	**destroy** ()
CameraSettings *	**getCameraSettings** ()
bool	**getCropping** (int *pOriginX, int *pOriginY, int *pWidth, int *pHeight) const
float	**getHorizontalFieldOfView** () const
int	**getMaxPixelValue** () const
int	**getMinPixelValue** () const
bool	**getMirroringEnabled** () const
Status	**getProperty** (int propertyId, void *data, int *dataSize) const
template<class T > **Status**	**getProperty** (int propertyId, T *value) const
const **SensorInfo** &	**getSensorInfo** () const
float	**getVerticalFieldOfView** () const
VideoMode	**getVideoMode** () const
Status	**invoke** (int commandId, void *data, int dataSize)
template<class T > **Status**	**invoke** (int commandId, T &value)
bool	**isCommandSupported** (int commandId) const
bool	**isCroppingSupported** () const
bool	**isPropertySupported** (int propertyId) const
bool	**isValid** () const
Status	**readFrame** (**VideoFrameRef** *pFrame)
void	**removeNewFrameListener** (**NewFrameListener** *pListener)

Status	**resetCropping** ()
Status	**setCropping** (int originX, int originY, int width, int height)
Status	**setFrameBuffersAllocator** (**FrameAllocator** *pAllocator)
Status	**setMirroringEnabled** (bool isEnabled)
Status	**setProperty** (int propertyId, const void *data, int dataSize)
template<class T >	
Status	**setProperty** (int propertyId, const T &value)
Status	**setVideoMode** (const **VideoMode** &videoMode)
Status	**start** ()
void	**stop** ()

Friends

class	**Device**

Detailed Description

The **VideoStream** object encapsulates a single video stream from a device. Once created, it is used to start data flow from the device, and to read individual frames of data. This is the central class used to obtain data in **OpenNI**. It provides the ability to manually read data in a polling loop, as well as providing events and a Listener class that can be used to implement event-driven data acquisition.

Aside from the video data frames themselves, the class offers a number of functions used for obtaining information about a **VideoStream**. Field of view, available video modes, and minimum and maximum valid pixel values can all be obtained.

In addition to obtaining data, the **VideoStream** object is used to set all configuration properties that apply to a specific stream (rather than to an entire device). In particular, it is used to control cropping, mirroring, and video modes.

A pointer to a valid, initialized device that provides the desired stream type is required to create a stream.

Several video streams can be created to stream data from the same sensor. This is useful if several components of an application need to read frames separately.

While some device might allow different streams from the same sensor to have different configurations, most devices will have a single configuration for the sensor, shared by all streams.

Constructor & Destructor Documentation

openni::VideoStream::VideoStream ( ) `[inline]`

Default constructor. Creates a new, non-valid **VideoStream** object. The object created will be invalid until its **create()** function is called with a valid **Device**.

openni::VideoStream::VideoStream (OniStreamHandle *handle*) `[i`

Handle constructor. Creates a **VideoStream** object based on the given initialized handle. This object will not destroy the underlying handle when **destroy()** or destructor is called

openni::VideoStream::~VideoStream ( ) `[inline]`

Destructor. The destructor calls the **destroy()** function, but it is considered a best practice for applications to call **destroy()** manually on any **VideoStream** that they run **create()** on.

Member Function Documentation

Status openni::VideoStream::addNewFrameListener (NewFrameLi

Adds a new Listener to receive this **VideoStream** onNewFrame event. See **VideoStream::NewFrameListener** for more information on implementing an event driven frame reading architecture. An instance of a listener can be added to only one source.

Parameters:

`[in]`	**pListener**	Pointer to a **VideoStream::NewFrameListener** object (or a derivative) that will respond to this event.

Returns:

Status code indicating success or failure of the operation.

Status openni::VideoStream::create (const Device & device, SensorType sensorType) [inline]

Creates a stream of frames from a specific sensor type of a specific device. You must supply a reference to a **Device** that supplies the sensor type requested. You can use **Device::hasSensor()** to check whether a given sensor is available on your target device before calling **create()**.

Parameters:

`[in]`	**device**	A reference to the **Device** you want to create the stream on.
`[in]`	**sensorType**	The type of sensor the stream should produce data from.

Returns:
Status code indicating success or failure for this operation.

void openni::VideoStream::destroy ( ) `[inline]`

Destroy this stream. This function is currently called automatically by the destructor, but it is considered a best practice for applications to manually call this function on any **VideoStream** that they call **create()** for.

CameraSettings* openni::VideoStream::getCameraSettings ( ) `[inl`

Gets an object through which several camera settings can be configured.

Returns:
NULL if the stream doesn't support camera settings.

bool openni::VideoStream::getCropping (int * pOriginX, int * pOriginY, int * pWidth, int * pHeight) const `[inline]`

Obtains the current cropping settings for this stream.

Parameters:

`[out]`	**pOriginX**	X coordinate of the upper left corner of the cropping window
`[out]`	**pOriginY**	Y coordinate of the upper left corner of the cropping window
`[out]`	**pWidth**	Horizontal width of the cropping window, in pixels

[out]	**pHeight**	Vertical width of the cropping window, in pixels returns true if cropping is currently enabled, false if it is not.

float openni::VideoStream::getHorizontalFieldOfView ( ) const `[in`

Gets the horizontal field of view of frames received from this stream.

Returns:
Horizontal field of view, in radians.

int openni::VideoStream::getMaxPixelValue ( ) const `[inline]`

Provides the maximum possible value for pixels obtained by this stream. This is most useful for getting the maximum possible value of depth streams.

Returns:
Maximum possible pixel value.

int openni::VideoStream::getMinPixelValue ( ) const `[inline]`

Provides the smallest possible value for pixels obtains by this **VideoStream**. This is most useful for getting the minimum possible value that will be reported by a depth stream.

Returns:
Minimum possible pixel value that can come from this stream.

bool openni::VideoStream::getMirroringEnabled ( ) const `[inline]`

Check whether mirroring is currently turned on for this stream.

Returns:
true if mirroring is currently enabled, false otherwise.

Status openni::VideoStream::getProperty (	int	**propertyId,**	
	void *	**data,**	
	int *	**dataSize**	
)		const `[inline]`	

General function for obtaining the value of stream specific properties. There are convenience functions available for all commonly used properties, so it is not expected that applications will make direct use of the getProperty function very often.

Parameters:

`[in]`	**propertyId**	The numerical ID of the property to be queried.
`[out]`	**data**	Place to store the value of the property.
`[in,out]`	**dataSize**	IN: Size of the buffer passed in the `data` argument. OUT: the actual written size.

Returns:
Status code indicating success or failure of this operation.

template<class T >

Status openni::VideoStream::getProperty (	int	**propertyId,**	
	T *	**value**	
)		const `[inline]`	

Function for getting the value from a property using an arbitrary output type. There are convenience functions available for all commonly used properties, so it is not expected that applications will make direct use of this function very often.

Template Parameters:

[in] T Data type of the value to be read.

Parameters:

`[in]`	**propertyId**	The numerical ID of the property to be read.
`[in,out]`	**value**	Pointer to a place to store the value read from the property.

Returns:

Status code indicating success or failure of this operation.

const SensorInfo& openni::VideoStream::getSensorInfo ( ) const `[`

Provides the **SensorInfo** object associated with the sensor that is producing this **VideoStream**. Note that this function will return NULL if the stream has not yet been initialized with the **create()** function.

SensorInfo is useful primarily as a means of learning which video modes are valid for this **VideoStream**.

Returns:

Reference to the **SensorInfo** object associated with the sensor providing this stream.

float openni::VideoStream::getVerticalFieldOfView ( ) const `[inlin`

Gets the vertical field of view of frames received from this stream.

Returns:

Vertical field of view, in radians.

VideoMode openni::VideoStream::getVideoMode ( ) const `[inline]`

Get the current video mode information for this video stream. This includes its resolution, fps and stream format.

Returns:
Current video mode information for this video stream.

Status openni::VideoStream::invoke (	int	commandId,
	void *	data,
	int	dataSize
)		[inline]

Invokes a command that takes an arbitrary data type as its input. It is not expected that application code will need this function frequently, as all commonly used properties have higher level functions provided.

Parameters:

[in]	**commandId**	Numerical code of the property to be invoked.
[in]	**data**	Data to be passed to the property.
[in]	**dataSize**	size of the buffer passed in `data`.

Returns:
Status code indicating success or failure of this operation.

template<class T >

Status openni::VideoStream::invoke (	int	commandId,
	T &	value
)		[inline]

Invokes a command that takes an arbitrary data type as its input. It is not expected that application code will need this function frequently, as all commonly used properties have higher level functions provided.

Template Parameters:

[in] T Type of data to be passed to the property.

Parameters:

[in]	**commandId**	Numerical code of the property to be invoked.
[in]	**value**	Data to be passed to the property.

Returns:

Status code indicating success or failure of this operation.

bool openni::VideoStream::isCommandSupported (int commandIc

Checks if a specific command is supported by the video stream.

Parameters:

[in]	**commandId**	Command to be checked.

Returns:

true if the command is supported, false otherwise.

bool openni::VideoStream::isCroppingSupported ( ) const [inline]

Checks whether this stream supports cropping.

Returns:

true if the stream supports cropping, false if it does not.

bool openni::VideoStream::isPropertySupported (int propertyId) c

Checks if a specific property is supported by the video stream.

Parameters:

[in]	**propertyId**	Property to be checked.

Returns:
true if the property is supported, false otherwise.

bool openni::VideoStream::isValid ( ) const `[inline]`

Checks to see if this object has been properly initialized and currently points to a valid stream.

Returns:
true if this object has been previously initialized, false otherwise.

Status openni::VideoStream::readFrame (VideoFrameRef * pFram

Read the next frame from this video stream, delivered as a **VideoFrameRef**. This is the primary method for manually obtaining frames of video data. If no new frame is available, the call will block until one is available. To avoid blocking, use VideoStream::Listener to implement an event driven architecture. Another alternative is to use **OpenNI::waitForAnyStream()** to wait for new frames from several streams.

Parameters:

`[out]`	**pFrame**	Pointer to a **VideoFrameRef** object to hold the reference to the new frame.

Returns:
Status code to indicated success or failure of this function.

void openni::VideoStream::removeNewFrameListener (NewFrameL

Removes a Listener from this video stream list. The listener removed will no longer receive new frame events from this stream.

Parameters:

`[in]`	**pListener**	Pointer to the listener object to be removed.

Status openni::VideoStream::resetCropping ( ) `[inline]`

Disables cropping.

Returns:
Status code indicating success or failure of this operation.

Status openni::VideoStream::setCropping (int originX, int originY, int width, int height) `[inline]`

Changes the cropping settings for this stream. You can use the isCroppingSupported() function to make sure cropping is supported before calling this function.

Parameters:

`[in]`	**originX**	New X coordinate of the upper left corner of the cropping window.
`[in]`	**originY**	New Y coordinate of the upper left corner of the cropping window.
`[in]`	**width**	New horizontal width for the cropping window, in pixels.
`[in]`	**height**	New vertical height for the cropping window, in pixels.

Returns:
Status code indicating success or failure of this operation.

Status openni::VideoStream::setFrameBuffersAllocator (FrameAll

Sets the frame buffers allocator for this video stream.

Parameters:

[in]	**pAllocator**	Pointer to the frame buffers allocator object. Pass NULL to return to default frame allocator.

Returns:

ONI_STATUS_OUT_OF_FLOW The frame buffers allocator cannot be set while stream is streaming.

Status **openni::VideoStream::setMirroringEnabled (bool isEnablec**

Enable or disable mirroring for this stream.

Parameters:

[in]	**isEnabled**	true to enable mirroring, false to disable it.

Returns:

Status code indicating the success or failure of this operation.

```
Status openni::VideoStream::setProperty ( int          propertyId,
                                          const void * data,
                                          int          dataSize
                                        )              [inline]
```

General function for setting the value of stream specific properties. There are convenience functions available for all commonly used properties, so it is not expected that applications will make direct use of the setProperty function very often.

Parameters:

[in]	**propertyId**	The numerical ID of the property to be set.
[in]	**data**	Place to store the data to be written to the

		property.
[in]	**dataSize**	Size of the data to be written to the property.

Returns:
Status code indicating success or failure of this operation.

template<class T >

Status openni::VideoStream::setProperty (int **propertyId,**
const T & **value**
) `[inline]`

Function for setting a value of a stream property using an arbitrary input type. There are convenience functions available for all commonly used properties, so it is not expected that applications will make direct use of this function very often.

Template Parameters:
[in] T Data type of the value to be passed to the property.

Parameters:

[in]	**propertyId**	The numerical ID of the property to be set.
[in]	**value**	Data to be sent to the property.

Returns:
Status code indicating success or failure of this operation.

Status openni::VideoStream::setVideoMode (const VideoMode &

Changes the current video mode of this stream. Recommended practice is to use **Device::getSensorInfo()**, and then **SensorInfo::getSupportedVideoModes()** to obtain a list of valid video mode settings for this stream. Then, pass a valid **VideoMode** to **setVideoMode** to ensure correct operation.

Parameters:

[in]	**videoMode**	Desired new video mode for this stream. returns Status code indicating success or failure of this operation.

Status openni::VideoStream::start ( ) [inline]

Starts data generation from this video stream.

void openni::VideoStream::stop ( ) [inline]

Stops data generation from this video stream.

Friends And Related Function Documentation

friend class Device `[friend]`

The documentation for this class was generated from the following file:

- **OpenNI.h**

openni::YUV422DoublePixel Struct Reference

`#include <OpenNI.h>`

List of all members.

Public Attributes

uint8_t	**u**
uint8_t	**v**
uint8_t	**y1**
uint8_t	**y2**

Detailed Description

Holds the value of two pixels in YUV422 format (Luminance/Chrominance,16-bits/pixel). The first pixel has the values y1, u, v. The second pixel has the values y2, u, v.

Member Data Documentation

uint8_t openni::YUV422DoublePixel::u

First chrominance value for two pixels, stored as blue luminance difference signal.

uint8_t openni::YUV422DoublePixel::v

Second chrominance value for two pixels, stored as red luminance difference signal.

uint8_t openni::YUV422DoublePixel::y1

Overall luminance value of first pixel.

uint8_t openni::YUV422DoublePixel::y2

Overall luminance value of second pixel.

The documentation for this struct was generated from the following file:

- **OpenNI.h**

Class Index

Here is a list of all class members with links to the classes they belong to:

- a -

- addDeviceConnectedListener() : **openni::OpenNI**
- addDeviceDisconnectedListener() : **openni::OpenNI**
- addDeviceStateChangedListener() : **openni::OpenNI**
- addNewFrameListener() : **openni::VideoStream**
- allocateFrameBuffer() : **openni::VideoStream::FrameAllocator**
- Array() : **openni::Array< T >**
- attach() : **openni::Recorder**

- b -

- b : **openni::RGB888Pixel**
- build : **openni::Version**

- c -

- close() : **openni::Device**
- convertDepthToColor() : **openni::CoordinateConverter**
- convertDepthToWorld() : **openni::CoordinateConverter**
- convertWorldToDepth() : **openni::CoordinateConverter**
- create() : **openni::Recorder** , **openni::VideoStream**

- d -

- destroy() : **openni::VideoStream** , **openni::Recorder**

- ~NewFrameListener() : **openni::VideoStream::NewFrameListener**
- ~PlaybackControl() : **openni::PlaybackControl**
- ~Recorder() : **openni::Recorder**
- ~VideoFrameRef() : **openni::VideoFrameRef**
- ~VideoStream() : **openni::VideoStream**

Namespace List

Here is a list of all namespaces with brief descriptions:

openni	

Classes | Typedefs | Enumerations | Variables

openni Namespace Reference

Classes

class	**Array**
class	**CameraSettings**
class	**CoordinateConverter**
class	**Device**
class	**DeviceInfo**
class	**OpenNI**
class	**PlaybackControl**
class	**Recorder**
struct	**RGB888Pixel**
class	**SensorInfo**
struct	**Version**
class	**VideoFrameRef**
class	**VideoMode**
class	**VideoStream**
struct	**YUV422DoublePixel**

Typedefs

typedef uint16_t	**DepthPixel**
typedef uint16_t	**Grayscale16Pixel**

Enumerations

enum	**DeviceState** { **DEVICE_STATE_OK** = 0, **DEVICE_STATE_ERROR** = 1, **DEVICE_STATE_NOT_READY** = 2, **DEVICE_STATE_EOF** = 3 }
enum	**ImageRegistrationMode** { **IMAGE_REGISTRATION_OFF** = 0, **IMAGE_REGISTRATION_DEPTH_TO_COLOR** = 1 }
enum	**PixelFormat** { **PIXEL_FORMAT_DEPTH_1_MM** = 100, **PIXEL_FORMAT_DEPTH_100_UM** = 101, **PIXEL_FORMAT_SHIFT_9_2** = 102, **PIXEL_FORMAT_SHIFT_9_3** = 103, **PIXEL_FORMAT_RGB888** = 200, **PIXEL_FORMAT_YUV422** = 201, **PIXEL_FORMAT_GRAY8** = 202, **PIXEL_FORMAT_GRAY16** = 203, **PIXEL_FORMAT_JPEG** = 204, **PIXEL_FORMAT_YUYV** = 205 }
enum	**SensorType** { **SENSOR_IR** = 1, **SENSOR_COLOR** = 2, **SENSOR_DEPTH** = 3 }
enum	**Status** { **STATUS_OK** = 0, **STATUS_ERROR** = 1, **STATUS_NOT_IMPLEMENTED** = 2, **STATUS_NOT_SUPPORTED** = 3, **STATUS_BAD_PARAMETER** = 4, **STATUS_OUT_OF_FLOW** = 5, **STATUS_NO_DEVICE** = 6, **STATUS_TIME_OUT** = 102 }

Variables

static const char *	**ANY_DEVICE** = NULL
static const int	**TIMEOUT_FOREVER** = -1
static const int	**TIMEOUT_NONE** = 0

Detailed Description

openni is the namespace of the entire C++ API of **OpenNI**

Typedef Documentation

typedef uint16_t openni::DepthPixel

Pixel type used to store depth images.

typedef uint16_t openni::Grayscale16Pixel

Pixel type used to store IR images.

Enumeration Type Documentation

enum openni::DeviceState

Enumerator:

DEVICE_STATE_OK

DEVICE_STATE_ERROR

DEVICE_STATE_NOT_READY

DEVICE_STATE_EOF

enum openni::ImageRegistrationMode

Enumerator:

IMAGE_REGISTRATION_OFF

IMAGE_REGISTRATION_DEPTH_TO_COLOR

enum openni::PixelFormat

All available formats of the output of a stream

Enumerator:

PIXEL_FORMAT_DEPTH_1_MM

PIXEL_FORMAT_DEPTH_100_UM

PIXEL_FORMAT_SHIFT_9_2

PIXEL_FORMAT_SHIFT_9_3

PIXEL_FORMAT_RGB888

PIXEL_FORMAT_YUV422

PIXEL_FORMAT_GRAY8

PIXEL_FORMAT_GRAY16

PIXEL_FORMAT_JPEG

PIXEL_FORMAT_YUYV

enum openni::SensorType

The source of the stream

Enumerator:

SENSOR_IR

SENSOR_COLOR

SENSOR_DEPTH

enum openni::Status

Possible failure values

Enumerator:

STATUS_OK

STATUS_ERROR

STATUS_NOT_IMPLEMENTED

STATUS_NOT_SUPPORTED

STATUS_BAD_PARAMETER

STATUS_OUT_OF_FLOW

STATUS_NO_DEVICE

STATUS_TIME_OUT

Variable Documentation

const char* openni::ANY_DEVICE = NULL `[static]`

This special URI can be passed to **Device::open()** when the application has no concern for a specific device.

const int openni::TIMEOUT_FOREVER = -1 `[static]`

const int openni::TIMEOUT_NONE = 0 `[static]`

Here is a list of all namespace members with links to the namespace documentation for each member:

- a -

- ANY_DEVICE : **openni**

- d -

- DepthPixel : **openni**
- DEVICE_STATE_EOF : **openni**
- DEVICE_STATE_ERROR : **openni**
- DEVICE_STATE_NOT_READY : **openni**
- DEVICE_STATE_OK : **openni**
- DeviceState : **openni**

- g -

- Grayscale16Pixel : **openni**

- i -

- IMAGE_REGISTRATION_DEPTH_TO_COLOR : **openni**
- IMAGE_REGISTRATION_OFF : **openni**
- ImageRegistrationMode : **openni**

- p -

- PIXEL_FORMAT_DEPTH_100_UM : **openni**
- PIXEL_FORMAT_DEPTH_1_MM : **openni**

- PIXEL_FORMAT_GRAY16 : **openni**
- PIXEL_FORMAT_GRAY8 : **openni**
- PIXEL_FORMAT_JPEG : **openni**
- PIXEL_FORMAT_RGB888 : **openni**
- PIXEL_FORMAT_SHIFT_9_2 : **openni**
- PIXEL_FORMAT_SHIFT_9_3 : **openni**
- PIXEL_FORMAT_YUV422 : **openni**
- PIXEL_FORMAT_YUYV : **openni**
- PixelFormat : **openni**

- s -

- SENSOR_COLOR : **openni**
- SENSOR_DEPTH : **openni**
- SENSOR_IR : **openni**
- SensorType : **openni**
- Status : **openni**
- STATUS_BAD_PARAMETER : **openni**
- STATUS_ERROR : **openni**
- STATUS_NO_DEVICE : **openni**
- STATUS_NOT_IMPLEMENTED : **openni**
- STATUS_NOT_SUPPORTED : **openni**
- STATUS_OK : **openni**
- STATUS_OUT_OF_FLOW : **openni**
- STATUS_TIME_OUT : **openni**

- t -

- TIMEOUT_FOREVER : **openni**
- TIMEOUT_NONE : **openni**

File List

Here is a list of all files with brief descriptions:

OniEnums.h [code]	
OpenNI.h [code]	

Namespaces | Enumerations | Variables

OniEnums.h File Reference

Go to the source code of this file.

Namespaces

namespace	**openni**

Enumerations

enum	**openni::DeviceState** { **openni::DEVICE_STATE_OK** = 0, **openni::DEVICE_STATE_ERROR** = 1, **openni::DEVICE_STATE_NOT_READY** = 2, **openni::DEVICE_STATE_EOF** = 3 }
enum	**openni::ImageRegistrationMode** { **openni::IMAGE_REGISTRATION_OFF** = 0, **openni::IMAGE_REGISTRATION_DEPTH_TO_COLOR** = 1 }
enum	**openni::PixelFormat** { **openni::PIXEL_FORMAT_DEPTH_1_MM** = 100, **openni::PIXEL_FORMAT_DEPTH_100_UM** = 101, **openni::PIXEL_FORMAT_SHIFT_9_2** = 102, **openni::PIXEL_FORMAT_SHIFT_9_3** = 103, **openni::PIXEL_FORMAT_RGB888** = 200, **openni::PIXEL_FORMAT_YUV422** = 201, **openni::PIXEL_FORMAT_GRAY8** = 202, **openni::PIXEL_FORMAT_GRAY16** = 203, **openni::PIXEL_FORMAT_JPEG** = 204, **openni::PIXEL_FORMAT_YUYV** = 205 }
enum	**openni::SensorType** { **openni::SENSOR_IR** = 1, **openni::SENSOR_COLOR** = 2, **openni::SENSOR_DEPTH** = 3 }
enum	**openni::Status** { **openni::STATUS_OK** = 0, **openni::STATUS_ERROR** = 1, **openni::STATUS_NOT_IMPLEMENTED** = 2, **openni::STATUS_NOT_SUPPORTED** = 3, **openni::STATUS_BAD_PARAMETER** = 4, **openni::STATUS_OUT_OF_FLOW** = 5, **openni::STATUS_NO_DEVICE** = 6, **openni::STATUS_TIME_OUT** = 102 }

Variables

static const int	**openni::TIMEOUT_FOREVER** = -1
static const int	**openni::TIMEOUT_NONE** = 0

Classes | Namespaces | Typedefs | Variables

OpenNI.h File Reference

```
#include "OniPlatform.h" #include "OniProperties.h"
#include "OniEnums.h"
#include "OniCAPI.h"
#include "OniCProperties.h"
```

Go to the source code of this file.

Classes

class	**openni::Array< T >**
class	**openni::CameraSettings**
class	**openni::CoordinateConverter**
class	**openni::Device**
class	**openni::OpenNI::DeviceConnectedListener**
class	**openni::OpenNI::DeviceDisconnectedListener**
class	**openni::DeviceInfo**
class	**openni::OpenNI::DeviceStateChangedListener**
class	**openni::VideoStream::FrameAllocator**
class	**openni::VideoStream::NewFrameListener**
class	**openni::OpenNI**
class	**openni::PlaybackControl**
class	**openni::Recorder**
struct	**openni::RGB888Pixel**
class	**openni::SensorInfo**
struct	**openni::Version**
class	**openni::VideoFrameRef**
class	**openni::VideoMode**
class	**openni::VideoStream**
struct	**openni::YUV422DoublePixel**

Namespaces

namespace	**openni**

Typedefs

typedef uint16_t	**openni::DepthPixel**
typedef uint16_t	**openni::Grayscale16Pixel**

Variables

static const char * **openni::ANY_DEVICE** = NULL

OpenNI.h

Go to the documentation of this file.

```
/*****************************************************************************
*                                                                            *
*  OpenNI 2.x Alpha                                                          *
*  Copyright (C) 2012 PrimeSense Ltd.                                        *
*                                                                            *
*  This file is part of OpenNI.                                              *
*                                                                            *
*  Licensed under the Apache License, Version 2.0 (the "License");          *
*  you may not use this file except in compliance with the License.          *
*  You may obtain a copy of the License at                                   *
*                                                                            *
*      http://www.apache.org/licenses/LICENSE-2.0                            *
*                                                                            *
*  Unless required by applicable law or agre
```

```
ed to in writing, software       *
*  distributed under the License is distribu
ted on an "AS IS" BASIS,         *
*  WITHOUT WARRANTIES OR CONDITIONS OF ANY K
IND, either express or implied.  *
*  See the License for the specific language
 governing permissions and       *
*  limitations under the License.
                                 *
*
                                 *
********************************************
*********************************/
#ifndef _OPENNI_H_
#define _OPENNI_H_

#include "OniPlatform.h"
#include "OniProperties.h"
#include "OniEnums.h"

#include "OniCAPI.h"
#include "OniCProperties.h"

namespace openni
{

typedef uint16_t                DepthPixel;

typedef uint16_t                Grayscale16P
ixel;

// structs
typedef struct
{
    int major;
    int minor;
    int maintenance;
    int build;
} Version;

typedef struct
{
    /* Red value of this pixel. */
    uint8_t r;
    /* Green value of this pixel. */
    uint8_t g;
    /* Blue value of this pixel. */
    uint8_t b;
} RGB888Pixel;

typedef struct
{
    uint8_t u;
    uint8_t y1;
    uint8_t v;
    uint8_t y2;
} YUV422DoublePixel;

#if ONI_PLATFORM != ONI_PLATFORM_WIN32
#pragma GCC diagnostic ignored "-Wunused-variable"
#pragma GCC diagnostic push
#endif
static const char* ANY_DEVICE = NULL;
#if ONI_PLATFORM != ONI_PLATFORM_WIN32
#pragma GCC diagnostic pop
#endif

template<class T>
class Array
{
public:
    Array() : m_data(NULL), m_count(0), m_owner(false) {}

    Array(const T* data, int count) : m_owner(false) { _setData(data, count); }

    ~Array()
    {
        clear();
    }

    int getSize() const { return m_count; }

    const T& operator[](int index) const {return m_data[index];}

    void _setData(const T* data, int count, bool isOwner = false)
    {
        clear();
        m_count = count;
        m_owner = isOwner;
        if (!isOwner)
        {
            m_data = data;
        }
        else
        {
            m_data = new T[count];
            memcpy((void*)m_data, data, count*sizeof(T));
        }
    }

private:
    Array(const Array<T>&);
    Array<T>& operator=(const Array<T>&);

    void clear()
    {
        if (m_owner && m_data != NULL)
            delete []m_data;
        m_owner = false;
        m_data = NULL;
        m_count = 0;
    }

    const T* m_data;
    int m_count;
    bool m_owner;
};

// Forward declaration of all
class SensorInfo;
class VideoStream;
class VideoFrameRef;
class Device;
class OpenNI;
class CameraSettings;
class PlaybackControl;

class VideoMode : private OniVideoMode
{
public:
    VideoMode()
    {}

    VideoMode(const VideoMode& other)
    {
        *this = other;
    }

    VideoMode& operator=(const VideoMode& other)
    {
        setPixelFormat(other.getPixelFormat(
));
        setResolution(other.getResolutionX(), other.getResolutionY());
        setFps(other.getFps());

        return *this;
    }

    PixelFormat getPixelFormat() const { return (PixelFormat)pixelFormat; }

    int getResolutionX() const { return resolutionX; }

    int getResolutionY() const {return resolutionY;}

    int getFps() const { return fps; }

    void setPixelFormat(PixelFormat format) { this->pixelFormat = (OniPixelFormat)format; }

    void setResolution(int resolutionX, int resolutionY)
    {
        this->resolutionX = resolutionX;
        this->resolutionY = resolutionY;
    }

    void setFps(int fps) { this->fps = fps; }

    friend class SensorInfo;
    friend class VideoStream;
    friend class VideoFrameRef;
};

```

```
class SensorInfo
{
public:
    SensorType getSensorType() const { return (SensorType)m_pInfo->sensorType; }

    const Array<VideoMode>& getSupportedVideoModes() const { return m_videoModes; }

private:
    SensorInfo(const SensorInfo&);
    SensorInfo& operator=(const SensorInfo&);

    SensorInfo() : m_pInfo(NULL), m_videoModes(NULL, 0) {}

    SensorInfo(const OniSensorInfo* pInfo) : m_pInfo(NULL), m_videoModes(NULL, 0)
    {
        _setInternal(pInfo);
    }

    void _setInternal(const OniSensorInfo* pInfo)
    {
        m_pInfo = pInfo;
        if (pInfo == NULL)
        {
            m_videoModes._setData(NULL, 0);
        }
        else
        {
            m_videoModes._setData(static_cast<VideoMode*>(pInfo->pSupportedVideoModes), pInfo->numSupportedVideoModes);
        }
    }

    const OniSensorInfo* m_pInfo;
    Array<VideoMode> m_videoModes;

    friend class VideoStream;
    friend class Device;
};

class DeviceInfo : private OniDeviceInfo
{
public:
    const char* getUri() const { return uri; }
    const char* getVendor() const { return vendor; }
    const char* getName() const { return name; }
    uint16_t getUsbVendorId() const { return usbVendorId; }
    uint16_t getUsbProductId() const { return usbProductId; }

    friend class Device;
    friend class OpenNI;
};

class VideoFrameRef
{
public:
    VideoFrameRef()
    {
        m_pFrame = NULL;
    }

    ~VideoFrameRef()
    {
        release();
    }

    VideoFrameRef(const VideoFrameRef& other) : m_pFrame(NULL)
    {
        _setFrame(other.m_pFrame);
    }

    VideoFrameRef& operator=(const VideoFrameRef& other)
    {
        _setFrame(other.m_pFrame);
        return *this;
    }

    inline int getDataSize() const
    {
        return m_pFrame->dataSize;
    }

    inline const void* getData() const
    {
        return m_pFrame->data;
    }

    inline SensorType getSensorType() const
    {
        return (SensorType)m_pFrame->sensorType;
    }

    inline const VideoMode& getVideoMode() const
    {
        return static_cast<const VideoMode&>(m_pFrame->videoMode);
    }

    inline uint64_t getTimestamp() const
    {
        return m_pFrame->timestamp;
    }

    inline int getFrameIndex() const
    {
        return m_pFrame->frameIndex;
    }

    inline int getWidth() const
    {
        return m_pFrame->width;
    }

    inline int getHeight() const
    {
        return m_pFrame->height;
    }

    inline bool getCroppingEnabled() const
    {
        return m_pFrame->croppingEnabled == TRUE;
    }

    inline int getCropOriginX() const
    {
        return m_pFrame->cropOriginX;
    }

    inline int getCropOriginY() const
    {
        return m_pFrame->cropOriginY;
    }
    inline int getStrideInBytes() const
    {
        return m_pFrame->stride;
    }

    inline bool isValid() const
    {
        return m_pFrame != NULL;
    }

    void release()
    {
        if (m_pFrame != NULL)
        {
            oniFrameRelease(m_pFrame);
            m_pFrame = NULL;
        }
    }

    void _setFrame(OniFrame* pFrame)
    {
        setReference(pFrame);
        if (pFrame != NULL)
        {
            oniFrameAddRef(pFrame);
        }
    }

    OniFrame* _getFrame()
    {
        return m_pFrame;
    }

private:
    friend class VideoStream;
    inline void setReference(OniFrame* pFram
e)
    {
        // Initial - don't addref. This is the reference from OpenNI
        release();
        m_pFrame = pFrame;
    }

    OniFrame* m_pFrame; // const!!?
};

class VideoStream
{
public:
    class NewFrameListener
    {
    public:
        NewFrameListener() : m_callbackHandle(NULL)
        {
        }

        virtual ~NewFrameListener()
        {
        }

        virtual void onNewFrame(VideoStream&) = 0;

    private:
        friend class VideoStream;

        static void ONI_CALLBACK_TYPE callback(OniStreamHandle streamHandle, void* pCookie)
        {
            NewFrameListener* pListener = (NewFrameListener*)pCookie;
            VideoStream stream;
            stream._setHandle(streamHandle);
            pListener->onNewFrame(stream);
            stream._setHandle(NULL);
        }
        OniCallbackHandle m_callbackHandle;
    };

    class FrameAllocator
    {
    public:
        virtual ~FrameAllocator() {}
        virtual void* allocateFrameBuffer(int size) = 0;
        virtual void freeFrameBuffer(void* data) = 0;

    private:
        friend class VideoStream;

        static void* ONI_CALLBACK_TYPE allocateFrameBufferCallback(int size, void* pCookie)
        {
            FrameAllocator* pThis = (FrameAllocator*)pCookie;
            return pThis->allocateFrameBuffer(size);
        }

        static void ONI_CALLBACK_TYPE freeFrameBufferCallback(void* data, void* pCookie)
        {
            FrameAllocator* pThis = (FrameAllocator*)pCookie;
            pThis->freeFrameBuffer(data);
        }
    };
```

```cpp

    VideoStream() : m_stream(NULL), m_sensorInfo(), m_pCameraSettings(NULL), m_isOwner(true)
    {}

    explicit VideoStream(OniStreamHandle handle) : m_stream(NULL), m_sensorInfo(), m_pCameraSettings(NULL), m_isOwner(false)
    {
        _setHandle(handle);
    }

    ~VideoStream()
    {
        destroy();
    }

    bool isValid() const
    {
        return m_stream != NULL;
    }

    inline Status create(const Device& device, SensorType sensorType);

    inline void destroy();

    const SensorInfo& getSensorInfo() const
    {
        return m_sensorInfo;
    }

    Status start()
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        return (Status)oniStreamStart(m_stream);
    }

    void stop()
    {
        if (!isValid())
        {
            return;
        }

        oniStreamStop(m_stream);
    }

    Status readFrame(VideoFrameRef* pFrame)
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        OniFrame* pOniFrame;
        Status rc = (Status)oniStreamReadFrame(m_stream, &pOniFrame);

        pFrame->setReference(pOniFrame);
        return rc;
    }

    Status addNewFrameListener(NewFrameListener* pListener)
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        return (Status)oniStreamRegisterNewFrameCallback(m_stream, pListener->callback, pListener, &pListener->m_callbackHandle);
    }

    void removeNewFrameListener(NewFrameListener* pListener)
    {
        if (!isValid())
        {
            return;
        }

        oniStreamUnregisterNewFrameCallback(m_stream, pListener->m_callbackHandle);
        pListener->m_callbackHandle = NULL;
    }

    Status setFrameBuffersAllocator(FrameAllocator* pAllocator)
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        if (pAllocator == NULL)
        {
            return (Status)oniStreamSetFrameBuffersAllocator(m_stream, NULL, NULL, NULL);
        }
        else
        {
            return (Status)oniStreamSetFrameBuffersAllocator(m_stream, pAllocator->allocateFra
meBufferCallback, pAllocator->freeFrameBufferCallback, pAllocator);
        }
    }

    OniStreamHandle _getHandle() const
    {
        return m_stream;
    }

    CameraSettings* getCameraSettings() {return m_pCameraSettings;}

    Status getProperty(int propertyId, void* data, int* dataSize) const
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        return (Status)oniStreamGetProperty(m_stream, propertyId, data, dataSize);
    }

    Status setProperty(int propertyId, const void* data, int dataSize)
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        return (Status)oniStreamSetProperty(m_stream, propertyId, data, dataSize);
    }

    VideoMode getVideoMode() const
    {
        VideoMode videoMode;
        getProperty<OniVideoMode>(STREAM_PROPERTY_VIDEO_MODE, static_cast<OniVideoMode*>(&videoMode));
        return videoMode;
    }

    Status setVideoMode(const VideoMode& videoMode)
    {
        return setProperty<OniVideoMode>(STREAM_PROPERTY_VIDEO_MODE, static_cast<const OniVideoMode&>(videoMode));
    }

    int getMaxPixelValue() const
    {
        int maxValue;
        Status rc = getProperty<int>(STREAM_PROPERTY_MAX_VALUE, &maxValue);
        if (rc != STATUS_OK)
        {
            return 0;
        }
        return maxValue;
    }

    int getMinPixelValue() const
    {
        int minValue;
        Status rc = getProperty<int>(STREAM_PROPERTY_MIN_VALUE, &minValue);
        if (rc != STATUS_OK)
        {
            return 0;
        }
        return minValue;
    }

    bool isCroppingSupported() const
    {
        return isPropertySupported(STREAM_PROPERTY_CROPPING);
    }

    bool getCropping(int* pOriginX, int* pOriginY, int* pWidth, int* pHeight) const
    {
        OniCropping cropping;
        bool enabled = false;

        Status rc = getProperty<OniCropping>(STREAM_PROPERTY_CROPPING, &cropping);

        if (rc == STATUS_OK)
        {
            *pOriginX = cropping.originX;
            *pOriginY = cropping.originY;
            *pWidth = cropping.width;
            *pHeight = cropping.height;
            enabled = (cropping.enabled == TRUE);
        }

        return enabled;
    }

    Status setCropping(int originX, int originY, int width, int height)
    {
        OniCropping cropping;
        cropping.enabled = true;
        cropping.originX = originX;
        cropping.originY = originY;
        cropping.width = width;
        cropping.height = height;
        return setProperty<OniCropping>(STREAM_PROPERTY_CROPPING, cropping);
    }

    Status resetCropping()
    {
        OniCropping cropping;
        cropping.enabled = false;
        return setProperty<OniCropping>(STREAM_PROPERTY_CROPPING, cropping);
    }

    bool getMirroringEnabled() const
    {
        OniBool enabled;
        Status rc = getProperty<OniBool>(STREAM_PROPERTY_MIRRORING, &enabled);
        if (rc != STATUS_OK)
        {
            return false;
        }
        return enabled == TRUE;
    }

    Status setMirroringEnabled(bool isEnabled)
    {
        return setProperty<OniBool>(STREAM_PROPERTY_MIRRORING, isEnabled ? TRUE : FALSE);
    }

    float getHorizontalFieldOfView() const
    {
        float horizontal = 0;
        getProperty<float>(STREAM_PROPERTY_HORIZONTAL_FOV, &horizontal);
        return horizontal;
    }

    float getVerticalFieldOfView() const
    {
        float vertical = 0;
        getProperty<float>(STREAM_PROPERTY_VERTICAL_FOV, &vertical);
        return vertical;
    }

    template <class T>
    Status setProperty(int propertyId, const T& value)
    {
        return setProperty(propertyId, &value, sizeof(T));
    }

    template <class T>
    Status getProperty(int propertyId, T* value) const
    {
        int size = sizeof(T);
        return getProperty(propertyId, value, &size);
    }

    bool isPropertySupported(int propertyId) const
    {
        if (!isValid())
        {
            return false;
        }

        return oniStreamIsPropertySupported(m_stream, propertyId) == TRUE;
    }

    Status invoke(int commandId, void* data, int dataSize)
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }

        return (Status)oniStreamInvoke(m_stream, commandId, data, dataSize);
    }

    template <class T>
    Status invoke(int commandId, T& value)
    {
        return invoke(commandId, &value, sizeof(T));
    }

    bool isCommandSupported(int commandId) const
    {
        if (!isValid())
        {
            return false;
        }

        return (Status)oniStreamIsCommandSupported(m_stream, commandId) == TRUE;
    }

```

```
private:
    friend class Device;

    void _setHandle(OniStreamHandle stream)
    {
        m_sensorInfo._setInternal(NULL);
        m_stream = stream;

        if (stream != NULL)
        {
            m_sensorInfo._setInternal(oniStreamGetSensorInfo(m_stream));
        }
    }

private:
    VideoStream(const VideoStream& other);
    VideoStream& operator=(const VideoStream& other);

    OniStreamHandle m_stream;
    SensorInfo m_sensorInfo;
    CameraSettings* m_pCameraSettings;
    bool m_isOwner;
};

class Device
{
public:
    Device() : m_pPlaybackControl(NULL), m_device(NULL), m_isOwner(true)
    {
        clearSensors();
    }

    explicit Device(OniDeviceHandle handle) : m_pPlaybackControl(NULL), m_device(NULL), m_isOw
```

```
ner(false)
    {
        _setHandle(handle);
    }

    ~Device()
    {
        if (m_device != NULL)
        {
            close();
        }
    }

    inline Status open(const char* uri);

    inline void close();

    const DeviceInfo& getDeviceInfo() const
    {
        return m_deviceInfo;
    }

    bool hasSensor(SensorType sensorType)
    {
        int i;
        for (i = 0; (i < ONI_MAX_SENSORS) && (m_aSensorInfo[i].m_pInfo != NULL); ++i)
        {
            if (m_aSensorInfo[i].getSensorType() == sensorType)
            {
                return true;
            }
        }

        if (i == ONI_MAX_SENSORS)
        {
            return false;
        }

        const OniSensorInfo* pInfo = oniDeviceGetSensorInfo(m_device, (OniSensorType)sensorType);

        if (pInfo == NULL)
        {
            return false;
        }

        m_aSensorInfo[i]._setInternal(pInfo);

        return true;
    }

    const SensorInfo* getSensorInfo(SensorType sensorType)
    {
        int i;
        for (i = 0; (i < ONI_MAX_SENSORS) && (m_aSensorInfo[i].m_pInfo != NULL); ++i)
        {
            if (m_aSensorInfo[i].getSensorType() == sensorType)
            {
                return &m_aSensorInfo[i];
            }
        }

        // not found. check to see we have additional space
        if (i == ONI_MAX_SENSORS)
        {
            return NULL;
        }

        const OniSensorInfo* pInfo = oniDeviceGetSensorInfo(m_device, (OniSensorType)sensorType);
        if (pInfo == NULL)
        {
            return NULL;
        }

        m_aSensorInfo[i]._setInternal(pInfo);
        return &m_aSensorInfo[i];
    }

    OniDeviceHandle _getHandle() const
    {
        return m_device;
    }

    PlaybackControl* getPlaybackControl() {return m_pPlaybackControl;}

    Status getProperty(int propertyId, void* data, int* dataSize) const
    {
        return (Status)oniDeviceGetProperty(m_device, propertyId, data, dataSize);
    }

    Status setProperty(int propertyId, const void* data, int dataSize)
    {
        return (Status)oniDeviceSetProperty(m_device, propertyId, data, dataSize);
    }

    bool isImageRegistrationModeSupported(ImageRegistrationMode mode) const
    {
        return (oniDeviceIsImageRegistrationModeSupported(m_device, (OniImageRegistrationMode)mode) == TRUE);
    }

    ImageRegistrationMode getImageRegistrationMode() const
    {
        ImageRegistrationMode mode;
        Status rc = getProperty<ImageRegistrationMode>(DEVICE_PROPERTY_IMAGE_REGISTRATION, &mode);
        if (rc != STATUS_OK)
        {
            return IMAGE_REGISTRATION_OFF;
        }
        return mode;
    }

    Status setImageRegistrationMode(ImageRegistrationMode mode)
    {
        return setProperty<ImageRegistrationMode>(DEVICE_PROPERTY_IMAGE_REGISTRATION, mode);
    }

    bool isValid() const
    {
        return m_device != NULL;
    }

    bool isFile() const
    {
        return isPropertySupported(DEVICE_PR
OPERTY_PLAYBACK_SPEED) &&
            isPropertySupported(DEVICE_PROPERTY_PLAYBACK_REPEAT_ENABLED) &&
            isCommandSupported(DEVICE_COMMAND_SEEK);
    }

    Status setDepthColorSyncEnabled(bool isEnabled)
    {
        Status rc = STATUS_OK;

        if (isEnabled)
        {
            rc = (Status)oniDeviceEnableDepthColorSync(m_device);
        }
        else
        {
            oniDeviceDisableDepthColorSync(m_device);
        }

        return rc;
    }

    bool getDepthColorSyncEnabled()
    {
        return oniDeviceGetDepthColorSyncEnabled(m_device) == TRUE;
    }

    template <class T>
    Status setProperty(int propertyId, const T& value)
    {
        return setProperty(propertyId, &valu
e, sizeof(T));
    }

    template <class T>
    Status getProperty(int propertyId, T* value) const
    {
        int size = sizeof(T);
        return getProperty(propertyId, value, &size);
    }

    bool isPropertySupported(int propertyId) const
    {
        return oniDeviceIsPropertySupported(m_device, propertyId) == TRUE;
    }

    Status invoke(int commandId, void* data, int dataSize)
    {
        return (Status)oniDeviceInvoke(m_device, commandId, data, dataSize);
    }

    template <class T>
    Status invoke(int propertyId, T& value)
    {
        return invoke(propertyId, &value, sizeof(T));
    }

    bool isCommandSupported(int commandId) const
    {
        return oniDeviceIsCommandSupported(m
_device, commandId) == TRUE;
    }

    inline Status _openEx(const char* uri, const char* mode);

private:
    Device(const Device&);
    Device& operator=(const Device&);

    void clearSensors()
    {
        for (int i = 0; i < ONI_MAX_SENSORS; ++i)
        {
            m_aSensorInfo[i]._setInternal(NULL);
        }
    }

    inline Status _setHandle(OniDeviceHandle deviceHandle);

private:
    PlaybackControl* m_pPlaybackControl;

    OniDeviceHandle m_device;
    DeviceInfo m_deviceInfo;
    SensorInfo m_aSensorInfo[ONI_MAX_SENSORS];

    bool m_isOwner;
};

class PlaybackControl
{
public:

    ~PlaybackControl()
    {
        detach();
    }

    float getSpeed() const
    {
        if (!isValid())
        {
            return 0.0f;
        }
        float speed;
        Status rc = m_pDevice->getProperty<float>(DEVICE_PROPERTY_PLAYBACK_SPEED, &speed);
        if (rc != STATUS_OK)
        {
            return 1.0f;
        }
        return speed;
    }
    Status setSpeed(float speed)
    {
        if (!isValid())
        {
            return STATUS_NO_DEVICE;
        }
        return m_pDevice->setProperty<float>(DEVICE_PROPERTY_PLAYBACK_SPEED, speed);
    }

    bool getRepeatEnabled() const
    {
        if (!isValid())
        {
            return false;
        }

        OniBool repeat;
        Status rc = m_pDevice->getProperty<OniBool>(DEVICE_PROPERTY_PLAYBACK_REPEAT_ENABLED, &repeat);
        if (rc != STATUS_OK)
        {
            return false;
        }

        return repeat == TRUE;
    }

    Status setRepeatEnabled(bool repeat)
    {
        if (!isValid())
        {
            return STATUS_NO_DEVICE;
        }

        return m_pDevice->setProperty<OniBool>(DEVICE_PROPERTY_PLAYBACK_REPEAT_ENABLED, repeat ? TRUE : FALSE);
    }

    Status seek(const VideoStream& stream, int frameIndex)
    {
        if (!isValid())
        {
            return STATUS_NO_DEVICE;
        }
        OniSeek seek;
        seek.frameIndex = frameIndex;
        seek.stream = stream._getHandle();
        return m_pDevice->invoke(DEVICE_COMMAND_SEEK, seek);
```

```cpp
    }

    int getNumberOfFrames(const VideoStream& stream) const
    {
        int numOfFrames = -1;
        Status rc = stream.getProperty<int>(STREAM_PROPERTY_NUMBER_OF_FRAMES, &numOfFrames);
        if (rc != STATUS_OK)
        {
            return 0;
        }
        return numOfFrames;
    }

    bool isValid() const
    {
        return m_pDevice != NULL;
    }
private:
    Status attach(Device* device)
    {
        if (!device->isValid() || !device->isFile())
        {
            return STATUS_ERROR;
        }

        detach();
        m_pDevice = device;

        return STATUS_OK;
    }
    void detach()
    {
        m_pDevice = NULL;
    }

    friend class Device;
    PlaybackControl(Device* pDevice) : m_pDevice(NULL)
    {
        if (pDevice != NULL)
        {
            attach(pDevice);
        }
    }

    Device* m_pDevice;
};

class CameraSettings
{
public:
    // setters
    Status setAutoExposureEnabled(bool enabled)
    {
        return setProperty(STREAM_PROPERTY_AUTO_EXPOSURE, enabled ? TRUE : FALSE);
    }
    Status setAutoWhiteBalanceEnabled(bool enabled)
    {
        return setProperty(STREAM_PROPERTY_AUTO_WHITE_BALANCE, enabled ? TRUE : FALSE);
    }

    bool getAutoExposureEnabled() const
    {
        OniBool enabled = FALSE;

        Status rc = getProperty(STREAM_PROPERTY_AUTO_EXPOSURE, &enabled);
        return rc == STATUS_OK && enabled == TRUE;
    }
    bool getAutoWhiteBalanceEnabled() const
    {
        OniBool enabled = FALSE;

        Status rc = getProperty(STREAM_PROPERTY_AUTO_WHITE_BALANCE, &enabled);
        return rc == STATUS_OK && enabled == TRUE;
    }

    Status setGain(int gain)
    {
        return setProperty(STREAM_PROPERTY_GAIN, gain);
    }
    Status setExposure(int exposure)
    {
        return setProperty(STREAM_PROPERTY_EXPOSURE, exposure);
    }
    int getGain()
    {
        int gain;
        Status rc = getProperty(STREAM_PROPERTY_GAIN, &gain);
        if (rc != STATUS_OK)
        {
            return 100;
        }
        return gain;
    }
    int getExposure()
    {
        int exposure;
        Status rc = getProperty(STREAM_PROPERTY_EXPOSURE, &exposure);
        if (rc != STATUS_OK)
        {
            return 0;
        }
        return exposure;
    }

    bool isValid() const {return m_pStream != NULL;}
private:
    template <class T>
    Status getProperty(int propertyId, T* value) const
    {
        if (!isValid()) return STATUS_NOT_SUPPORTED;

        return m_pStream->getProperty<T>(propertyId, value);
    }
    template <class T>
    Status setProperty(int propertyId, const T& value)
    {
        if (!isValid()) return STATUS_NOT_SUPPORTED;

        return m_pStream->setProperty<T>(propertyId, value);
    }

    friend class VideoStream;
    CameraSettings(VideoStream* pStream)
    {
        m_pStream = pStream;
    }

    VideoStream* m_pStream;
};


class OpenNI
{
public:

    class DeviceConnectedListener
    {
    public:
        DeviceConnectedListener()
        {
            m_deviceConnectedCallbacks.deviceConnected = deviceConnectedCallback;
            m_deviceConnectedCallbacks.deviceDisconnected = NULL;
            m_deviceConnectedCallbacks.deviceStateChanged = NULL;
            m_deviceConnectedCallbacksHandle = NULL;
        }

        virtual ~DeviceConnectedListener()
        {
        }

        virtual void onDeviceConnected(const DeviceInfo*) = 0;
    private:
        static void ONI_CALLBACK_TYPE deviceConnectedCallback(const OniDeviceInfo* pInfo, void* pCookie)
        {
            DeviceConnectedListener* pListen
```

```
er = (DeviceConnectedListener*)pCookie;
            pListener->onDeviceConnected(static_cast<const DeviceInfo*>(pInfo));
        }

        friend class OpenNI;
        OniDeviceCallbacks m_deviceConnectedCallbacks;
        OniCallbackHandle m_deviceConnectedCallbacksHandle;

    };
    class DeviceDisconnectedListener
    {
    public:
        DeviceDisconnectedListener()
        {
            m_deviceDisconnectedCallbacks.deviceConnected = NULL;
            m_deviceDisconnectedCallbacks.deviceDisconnected = deviceDisconnectedCallback;
            m_deviceDisconnectedCallbacks.deviceStateChanged = NULL;
            m_deviceDisconnectedCallbacksHandle = NULL;
        }

        virtual ~DeviceDisconnectedListener()
        {
        }

        virtual void onDeviceDisconnected(const DeviceInfo*) = 0;
    private:
        static void ONI_CALLBACK_TYPE deviceDisconnectedCallback(const OniDeviceInfo* pInfo, v
oid* pCookie)
        {
            DeviceDisconnectedListener* pListener = (DeviceDisconnectedListener*)pCookie;
            pListener->onDeviceDisconnected(static_cast<const DeviceInfo*>(pInfo));
        }

        friend class OpenNI;
        OniDeviceCallbacks m_deviceDisconnectedCallbacks;
        OniCallbackHandle m_deviceDisconnectedCallbacksHandle;
    };
    class DeviceStateChangedListener
    {
    public:
        DeviceStateChangedListener()
        {
            m_deviceStateChangedCallbacks.deviceConnected = NULL;
            m_deviceStateChangedCallbacks.deviceDisconnected = NULL;
            m_deviceStateChangedCallbacks.deviceStateChanged = deviceStateChangedCallback;
            m_deviceStateChangedCallbacksHandle = NULL;
        }

        virtual ~DeviceStateChangedListener()
        {
        }

        virtual void onDeviceStateChanged(const DeviceInfo*, DeviceState) = 0;
    private:
        static void ONI_CALLBACK_TYPE deviceStateChangedCallback(const OniDeviceInfo* pInfo, OniDeviceState state, void* pCookie)
        {
            DeviceStateChangedListener* pListener = (DeviceStateChangedListener*)pCookie;
            pListener->onDeviceStateChanged(static_cast<const DeviceInfo*>(pInfo), DeviceState(state));
        }

        friend class OpenNI;
        OniDeviceCallbacks m_deviceStateChangedCallbacks;
        OniCallbackHandle m_deviceStateChangedCallbacksHandle;
    };

    static Status initialize()
    {
        return (Status)oniInitialize(ONI_API_VERSION); // provide version of API, to make sure proper struct sizes are used
    }

    static void shutdown()
    {
        oniShutdown();
    }

    static Version getVersion()
    {
        OniVersion oniVersion = oniGetVersion();
        Version version;
        version.major = oniVersion.major;
        version.minor = oniVersion.minor;
        version.maintenance = oniVersion.maintenance;
        version.build = oniVersion.build;
        return version;
    }

    static const char* getExtendedError()
    {
        return oniGetExtendedError();
    }

    static void enumerateDevices(Array<DeviceInfo>* deviceInfoList)
    {
        OniDeviceInfo* m_pDeviceInfos;
        int m_deviceInfoCount;
        oniGetDeviceList(&m_pDeviceInfos, &m_deviceInfoCount);
        deviceInfoList->_setData((DeviceInfo*)m_pDeviceInfos, m_deviceInfoCount, true);
        oniReleaseDeviceList(m_pDeviceInfos);
    }

    static Status waitForAnyStream(VideoStream** pStreams, int streamCount, int* pReadyStreamIndex, int timeout = TIMEOUT_FOREVER)
    {
        static const int ONI_MAX_STREAMS = 50;
        OniStreamHandle streams[ONI_MAX_STREAMS];

        if (streamCount > ONI_MAX_STREAMS)
        {
            printf("Too many streams for wait: %d > %d\n", streamCount, ONI_MAX_STREAMS);
            return STATUS_BAD_PARAMETER;
        }

        *pReadyStreamIndex = -1;
        for (int i = 0; i < streamCount; ++i)
        {
            if (pStreams[i] != NULL)
            {
                streams[i] = pStreams[i]->_getHandle();
            }
            else
            {
                streams[i] = NULL;
            }
        }
        Status rc = (Status)oniWaitForAnyStream(streams, streamCount, pReadyStreamIndex, timeout);

        return rc;
    }

    static Status addDeviceConnectedListener(DeviceConnectedListener* pListener)
    {
        if (pListener->m_deviceConnectedCallbacksHandle != NULL)
        {
            return STATUS_ERROR;
        }
        return (Status)oniRegisterDeviceCallbacks(&pListener->m_deviceConnectedCallbacks, pListener, &pListener->m_deviceConnectedCallbacksHandle);
    }
    static Status addDeviceDisconnectedListener(DeviceDisconnectedListener* pListener)
    {
        if (pListener->m_deviceDisconnectedCallbacksHandle != NULL)
        {
            return STATUS_ERROR;
        }
        return (Status)oniRegisterDeviceCallbacks(&pListener->m_deviceDisconnectedCallbacks, pListener, &pListener->m_deviceDisconnectedCallbacksHandle);
    }
    static Status addDeviceStateChangedListener(DeviceStateChangedListener* pListener)
    {
        if (pListener->m_deviceStateChangedCallbacksHandle != NULL)
        {
            return STATUS_ERROR;
        }
        return (Status)oniRegisterDeviceCallbacks(&pListener->m_deviceStateChangedCallbacks, pListener, &pListener->m_deviceStateChangedCallbacksHandle);
    }
    static void removeDeviceConnectedListener(DeviceConnectedListener* pListener)
    {
        oniUnregisterDeviceCallbacks(pListener->m_deviceConnectedCallbacksHandle);
        pListener->m_deviceConnectedCallbacksHandle = NULL;
    }
    static void removeDeviceDisconnectedListener(DeviceDisconnectedListener* pListener)
    {
        oniUnregisterDeviceCallbacks(pListener->m_deviceDisconnectedCallbacksHandle);
        pListener->m_deviceDisconnectedCallbacksHandle = NULL;
    }
    static void removeDeviceStateChangedListener(DeviceStateChangedListener* pListener)
    {
        oniUnregisterDeviceCallbacks(pListener->m_deviceStateChangedCallbacksHandle);
        pListener->m_deviceStateChangedCallbacksHandle = NULL;
    }

    static Status setLogOutputFolder(const char *strLogOutputFolder)
    {
        return (Status)oniSetLogOutputFolder(strLogOutputFolder);
    }

    static Status getLogFileName(char *strFileName, int nBufferSize)
    {
        return (Status)oniGetLogFileName(strFileName, nBufferSize);
    }

    static Status setLogMinSeverity(int nMinSeverity)
    {
        return(Status) oniSetLogMinSeverity(nMinSeverity);
    }

    static Status setLogConsoleOutput(bool bConsoleOutput)
    {
        return (Status)oniSetLogConsoleOutput(bConsoleOutput);
    }

    static Status setLogFileOutput(bool bFileOutput)
    {
        return (Status)oniSetLogFileOutput(bFileOutput);
    }

    #if ONI_PLATFORM == ONI_PLATFORM_ANDROID_ARM

    static Status setLogAndroidOutput(bool bAndroidOutput)
    {
        return (Status)oniSetLogAndroidOutput(bAndroidOutput);
    }
    #endif

private:
    OpenNI()
    {
    }
};

class CoordinateConverter
{
public:
    static Status convertWorldToDepth(const VideoStream& depthStream, float worldX, float worldY, float worldZ, int* pDepthX, int* pDepthY, DepthPixel* pDepthZ)
    {
        float depthX, depthY, depthZ;
        Status rc = (Status)oniCoordinateConverterWorldToDepth(depthStream._getHandle(), worldX, worldY, worldZ, &depthX, &depthY, &depthZ);
        *pDepthX = (int)depthX;
        *pDepthY = (int)depthY;
        *pDepthZ = (DepthPixel)depthZ;
        return rc;
    }

    static Status convertWorldToDepth(const VideoStream& depthStream, float worldX, float worldY, float worldZ, float* pDepthX, float* pDepthY, float* pDepthZ)
    {
        return (Status)oniCoordinateConverterWorldToDepth(depthStream._getHandle(), worldX, worldY, worldZ, pDepthX, pDepthY, pDepthZ);
    }

    static Status convertDepthToWorld(const VideoStream& depthStream, int depthX, int depthY, DepthPixel depthZ, float* pWorldX, float* pWorldY, float* pWorldZ)
    {
        return (Status)oniCoordinateConverterDepthToWorld(depthStream._getHandle(), float(depthX), float(depthY), float(depthZ), pWorldX, pWorldY, pWorldZ);
    }

    static Status convertDepthToWorld(const VideoStream& depthStream, float depthX, float depthY, float depthZ, float* pWorldX, float* pWorldY, float* pWorldZ)
    {
        return (Status)oniCoordinateConverte
```

```
rDepthToWorld(depthStream._getHandle(), depthX, depthY, depthZ, pWorldX, pWorldY, pWorldZ);
    }

    static Status convertDepthToColor(const VideoStream& depthStream, const VideoStream& colorStream, int depthX, int depthY, DepthPixel depthZ, int* pColorX, int* pColorY)
    {
        return (Status)oniCoordinateConverterDepthToColor(depthStream._getHandle(), colorStream._getHandle(), depthX, depthY, depthZ, pColorX, pColorY);
    }
};

class Recorder
{
public:
    Recorder() : m_recorder(NULL)
    {
    }

    ~Recorder()
    {
        destroy();
    }

    Status create(const char* fileName)
    {
        if (!isValid())
        {
            return (Status)oniCreateRecorder(fileName, &m_recorder);
        }
        return STATUS_ERROR;
    }

    bool isValid() const
    {
        return NULL != getHandle();
    }

    Status attach(VideoStream& stream, bool allowLossyCompression = false)
    {
        if (!isValid() || !stream.isValid())
        {
            return STATUS_ERROR;
        }
        return (Status)oniRecorderAttachStream(
                m_recorder,
                stream._getHandle(),
                allowLossyCompression);
    }

    Status start()
    {
        if (!isValid())
        {
            return STATUS_ERROR;
        }
        return (Status)oniRecorderStart(m_recorder);
    }

    void stop()
    {
        if (isValid())
        {
            oniRecorderStop(m_recorder);
        }
    }

    void destroy()
    {
        if (isValid())
        {
            oniRecorderDestroy(&m_recorder);
        }
    }

private:
    Recorder(const Recorder&);
    Recorder& operator=(const Recorder&);

    OniRecorderHandle getHandle() const
    {
        return m_recorder;
    }


    OniRecorderHandle m_recorder;
};

// Implemetation
Status VideoStream::create(const Device& device, SensorType sensorType)
{
    OniStreamHandle streamHandle;
    Status rc = (Status)oniDeviceCreateStream(device._getHandle(), (OniSensorType)sensorType, &streamHandle);
    if (rc != STATUS_OK)
    {
        return rc;
    }

    m_isOwner = true;
    _setHandle(streamHandle);

    if (isPropertySupported(STREAM_PROPERTY_AUTO_WHITE_BALANCE) && isPropertySupported(STREAM_PROPERTY_AUTO_EXPOSURE))
    {
        m_pCameraSettings = new CameraSettings(this);
    }

    return STATUS_OK;
}

void VideoStream::destroy()
{
    if (!isValid())
    {
        return;
    }

    if (m_pCameraSettings != NULL)
    {
        delete m_pCameraSettings;
        m_pCameraSettings = NULL;
    }

    if (m_stream != NULL)
    {
        if(m_isOwner)
            oniStreamDestroy(m_stream);
        m_stream = NULL;
    }
}

Status Device::open(const char* uri)
{
    //If we are not the owners, we stick with our own device
    if(!m_isOwner)
    {
        if(isValid()){
            return STATUS_OK;
        }else{
            return STATUS_OUT_OF_FLOW;
        }
    }

    OniDeviceHandle deviceHandle;
    Status rc = (Status)oniDeviceOpen(uri, &deviceHandle);
    if (rc != STATUS_OK)
    {
        return rc;
    }

    _setHandle(deviceHandle);

    return STATUS_OK;
}

Status Device::_openEx(const char* uri, const char* mode)
{
    //If we are not the owners, we stick with our own device
    if(!m_isOwner)
    {
        if(isValid()){
            return STATUS_OK;
        }else{
            return STATUS_OUT_OF_FLOW;
        }
    }

    OniDeviceHandle deviceHandle;
    Status rc = (Status)oniDeviceOpenEx(uri, mode, &deviceHandle);
    if (rc != STATUS_OK)
    {
        return rc;
    }

    _setHandle(deviceHandle);

    return STATUS_OK;
}

Status Device::_setHandle(OniDeviceHandle deviceHandle)
{
    if (m_device == NULL)
    {
        m_device = deviceHandle;

        clearSensors();

        oniDeviceGetInfo(m_device, &m_deviceInfo);

        if (isFile())
        {
            m_pPlaybackControl = new PlaybackControl(this);
        }

        // Read deviceInfo
        return STATUS_OK;
    }

    return STATUS_OUT_OF_FLOW;
}

```

```
void Device::close()
{
    if (m_pPlaybackControl != NULL)
    {
        delete m_pPlaybackControl;
        m_pPlaybackControl = NULL;
    }

    if (m_device != NULL)
    {
        if(m_isOwner)
        {
            oniDeviceClose(m_device);
        }

        m_device = NULL;
    }
}


}

#endif // _OPEN_NI_HPP_
```

openni::Array< T > Member List

This is the complete list of members for **openni::Array< T >**, including all inherited members.

Array()	**openni::Array< T >**	`[inline]`
Array(const T *data, int count)	**openni::Array< T >**	`[inline]`
getSize() const	**openni::Array< T >**	`[inline]`
operator[](int index) const	**openni::Array< T >**	`[inline]`
~Array()	**openni::Array< T >**	`[inline]`

openni::CameraSettings Member List

This is the complete list of members for **openni::CameraSettings**, including all inherited members.

getAutoExposureEnabled() const	openni::CameraSettings	[inline
getAutoWhiteBalanceEnabled() const	openni::CameraSettings	[inline
getExposure()	openni::CameraSettings	[inline
getGain()	openni::CameraSettings	[inline
isValid() const	openni::CameraSettings	[inline
setAutoExposureEnabled(bool enabled)	openni::CameraSettings	[inline
setAutoWhiteBalanceEnabled(bool enabled)	openni::CameraSettings	[inline
setExposure(int exposure)	openni::CameraSettings	[inline
setGain(int gain)	openni::CameraSettings	[inline
VideoStream class	openni::CameraSettings	[friend

openni::CoordinateConverter Member List

This is the complete list of members for openni::CoordinateConverter, including all inherited members.

convertDepthToColor(const VideoStream &depthStream, const VideoStream &colorStream, int depthX, int depthY, DepthPixel depthZ, int *pColorX, int *pColorY)	openni::CoordinateConverter	[inline, static]
convertDepthToWorld(const VideoStream &depthStream, int depthX, int depthY, DepthPixel depthZ, float *pWorldX, float *pWorldY, float *pWorldZ)	openni::CoordinateConverter	[inline, static]
convertDepthToWorld(const VideoStream &depthStream, float depthX, float depthY, float depthZ, float *pWorldX, float *pWorldY, float *pWorldZ)	openni::CoordinateConverter	[inline, static]
convertWorldToDepth(const VideoStream &depthStream, float worldX, float worldY, float worldZ, int *pDepthX, int *pDepthY, DepthPixel *pDepthZ)	openni::CoordinateConverter	[inline, static]

convertWorldToDepth(const VideoStream &depthStream, float worldX, float worldY, float worldZ, float *pDepthX, float *pDepthY, float *pDepthZ)	**openni::CoordinateConverter**	`[inline, static]`

openni::Device Member List

This is the complete list of members for **openni::Device**, including all inherited members.

close()	**openni:**
Device()	**openni:**
Device(OniDeviceHandle handle)	**openni:**
getDepthColorSyncEnabled()	**openni:**
getDeviceInfo() const	**openni:**
getImageRegistrationMode() const	**openni:**
getPlaybackControl()	**openni:**
getProperty(int propertyId, void *data, int *dataSize) const	**openni:**
getProperty(int propertyId, T *value) const	**openni:**
getSensorInfo(SensorType sensorType)	**openni:**
hasSensor(SensorType sensorType)	**openni:**
invoke(int commandId, void *data, int dataSize)	**openni:**
invoke(int propertyId, T &value)	**openni:**
isCommandSupported(int commandId) const	**openni:**
isFile() const	**openni:**
isImageRegistrationModeSupported(ImageRegistrationMode mode) const	**openni:**
isPropertySupported(int propertyId) const	**openni:**
isValid() const	**openni:**
open(const char *uri)	**openni:**
setDepthColorSyncEnabled(bool isEnabled)	**openni:**

setImageRegistrationMode(ImageRegistrationMode mode)	**openni:**
setProperty(int propertyId, const void *data, int dataSize)	**openni:**
setProperty(int propertyId, const T &value)	**openni:**
~Device()	**openni:**

openni::OpenNI::DeviceConnectedListener Member List

This is the complete list of members for **openni::OpenNI::DeviceConnectedListener**, including all inherited members.

DeviceConnectedListener()	openni::OpenNI::DeviceConnectedList
onDeviceConnected(const DeviceInfo *)=0	openni::OpenNI::DeviceConnectedList
OpenNI class	openni::OpenNI::DeviceConnectedList
~DeviceConnectedListener()	openni::OpenNI::DeviceConnectedList

openni::OpenNI::DeviceDisconnectedListener Member List

This is the complete list of members for **openni::OpenNI::DeviceDisconnectedListener**, including all inherited members.

DeviceDisconnectedListener()	**openni::OpenNI::DeviceDisconnect**
onDeviceDisconnected(const DeviceInfo *)=0	**openni::OpenNI::DeviceDisconnect**
OpenNI class	**openni::OpenNI::DeviceDisconnect**
~DeviceDisconnectedListener()	**openni::OpenNI::DeviceDisconnect**

openni::DeviceInfo Member List

This is the complete list of members for **openni::DeviceInfo**, including all inherited members.

Device class	**openni::DeviceInfo**	`[friend]`
getName() const	**openni::DeviceInfo**	`[inline]`
getUri() const	**openni::DeviceInfo**	`[inline]`
getUsbProductId() const	**openni::DeviceInfo**	`[inline]`
getUsbVendorId() const	**openni::DeviceInfo**	`[inline]`
getVendor() const	**openni::DeviceInfo**	`[inline]`
OpenNI class	**openni::DeviceInfo**	`[friend]`

openni::OpenNI::DeviceStateChangedListener Member List

This is the complete list of members for **openni::OpenNI::DeviceStateChangedListener**, including all inherited members.

DeviceStateChangedListener()	openni::OpenNI::DeviceStateChang
onDeviceStateChanged(const DeviceInfo *, DeviceState)=0	openni::OpenNI::DeviceStateChang
OpenNI class	openni::OpenNI::DeviceStateChang
~DeviceStateChangedListener()	openni::OpenNI::DeviceStateChang

openni::VideoStream::FrameAllocator Member List

This is the complete list of members for **openni::VideoStream::FrameAllocator**, including all inherited members.

allocateFrameBuffer(int size)=0	**openni::VideoStream::FrameAllocator**	`[pure virtu`
freeFrameBuffer(void *data)=0	**openni::VideoStream::FrameAllocator**	`[pure virtu`
VideoStream class	**openni::VideoStream::FrameAllocator**	`[frie`
~FrameAllocator()	**openni::VideoStream::FrameAllocator**	`[inli virtu`

openni::VideoStream::NewFrameListener Member List

This is the complete list of members for **openni::VideoStream::NewFrameListener**, including all inherited members.

NewFrameListener()	openni::VideoStream::NewFrameListene
onNewFrame(VideoStream &)=0	openni::VideoStream::NewFrameListene
VideoStream class	openni::VideoStream::NewFrameListene
~NewFrameListener()	openni::VideoStream::NewFrameListene

openni::OpenNI Member List

This is the complete list of members for **openni::OpenNI**, including all inherited members.

addDeviceConnectedListener(DeviceConnectedListener *pListener)	**op**
addDeviceDisconnectedListener(DeviceDisconnectedListener *pListener)	**op**
addDeviceStateChangedListener(DeviceStateChangedListener *pListener)	**op**
enumerateDevices(Array< DeviceInfo > *deviceInfoList)	**op**
getExtendedError()	**op**
getLogFileName(char *strFileName, int nBufferSize)	**op**
getVersion()	**op**
initialize()	**op**
removeDeviceConnectedListener(DeviceConnectedListener *pListener)	**op**
removeDeviceDisconnectedListener(DeviceDisconnectedListener *pListener)	**op**
removeDeviceStateChangedListener(DeviceStateChangedListener *pListener)	**op**
setLogAndroidOutput(bool bAndroidOutput)	**op**
setLogConsoleOutput(bool bConsoleOutput)	**op**

setLogFileOutput(bool bFileOutput)	**op**
setLogMinSeverity(int nMinSeverity)	**op**
setLogOutputFolder(const char *strLogOutputFolder)	**op**
shutdown()	**op**
waitForAnyStream(VideoStream **pStreams, int streamCount, int *pReadyStreamIndex, int timeout=TIMEOUT_FOREVER)	**op**

openni::PlaybackControl Member List

This is the complete list of members for **openni::PlaybackControl**, including all inherited members.

Device class	**openni::PlaybackControl**	`[friend]`
getNumberOfFrames(const VideoStream &stream) const	**openni::PlaybackControl**	`[inline]`
getRepeatEnabled() const	**openni::PlaybackControl**	`[inline]`
getSpeed() const	**openni::PlaybackControl**	`[inline]`
isValid() const	**openni::PlaybackControl**	`[inline]`
seek(const VideoStream &stream, int frameIndex)	**openni::PlaybackControl**	`[inline]`
setRepeatEnabled(bool repeat)	**openni::PlaybackControl**	`[inline]`
setSpeed(float speed)	**openni::PlaybackControl**	`[inline]`
~PlaybackControl()	**openni::PlaybackControl**	`[inline]`

openni::Recorder Member List

This is the complete list of members for openni::Recorder, including all inherited members.

attach(VideoStream &stream, bool allowLossyCompression=false)	openni::Recorder	[inline]
create(const char *fileName)	openni::Recorder	[inline]
destroy()	openni::Recorder	[inline]
isValid() const	openni::Recorder	[inline]
Recorder()	openni::Recorder	[inline]
start()	openni::Recorder	[inline]
stop()	openni::Recorder	[inline]
~Recorder()	openni::Recorder	[inline]

openni::RGB888Pixel Member List

This is the complete list of members for **openni::RGB888Pixel**, including all inherited members.

b	openni::RGB888Pixel
g	openni::RGB888Pixel
r	openni::RGB888Pixel

openni::SensorInfo Member List

This is the complete list of members for **openni::SensorInfo**, including all inherited members.

Device class	openni::SensorInfo	[friend]
getSensorType() const	openni::SensorInfo	[inline]
getSupportedVideoModes() const	openni::SensorInfo	[inline]
VideoStream class	openni::SensorInfo	[friend]

openni::Version Member List

This is the complete list of members for **openni::Version**, including all inherited members.

build	openni::Version
maintenance	openni::Version
major	openni::Version
minor	openni::Version

openni::VideoFrameRef Member List

This is the complete list of members for **openni::VideoFrameRef**, including all inherited members.

getCropOriginX() const	openni::VideoFrameRef	[inline]
getCropOriginY() const	openni::VideoFrameRef	[inline]
getCroppingEnabled() const	openni::VideoFrameRef	[inline]
getData() const	openni::VideoFrameRef	[inline]
getDataSize() const	openni::VideoFrameRef	[inline]
getFrameIndex() const	openni::VideoFrameRef	[inline]
getHeight() const	openni::VideoFrameRef	[inline]
getSensorType() const	openni::VideoFrameRef	[inline]
getStrideInBytes() const	openni::VideoFrameRef	[inline]
getTimestamp() const	openni::VideoFrameRef	[inline]
getVideoMode() const	openni::VideoFrameRef	[inline]
getWidth() const	openni::VideoFrameRef	[inline]
isValid() const	openni::VideoFrameRef	[inline]
operator=(const VideoFrameRef &other)	openni::VideoFrameRef	[inline]
release()	openni::VideoFrameRef	[inline]
VideoFrameRef()	openni::VideoFrameRef	[inline]
VideoFrameRef(const VideoFrameRef &other)	openni::VideoFrameRef	[inline]
VideoStream class	openni::VideoFrameRef	[friend]
~VideoFrameRef()	openni::VideoFrameRef	[inline]

openni::VideoMode Member List

This is the complete list of members for **openni::VideoMode**, including all inherited members.

getFps() const	**openni::VideoMode**	`[inline]`
getPixelFormat() const	**openni::VideoMode**	`[inline]`
getResolutionX() const	**openni::VideoMode**	`[inline]`
getResolutionY() const	**openni::VideoMode**	`[inline]`
operator=(const VideoMode &other)	**openni::VideoMode**	`[inline]`
SensorInfo class	**openni::VideoMode**	`[friend]`
setFps(int fps)	**openni::VideoMode**	`[inline]`
setPixelFormat(PixelFormat format)	**openni::VideoMode**	`[inline]`
setResolution(int resolutionX, int resolutionY)	**openni::VideoMode**	`[inline]`
VideoFrameRef class	**openni::VideoMode**	`[friend]`
VideoMode()	**openni::VideoMode**	`[inline]`
VideoMode(const VideoMode &other)	**openni::VideoMode**	`[inline]`
VideoStream class	**openni::VideoMode**	`[friend]`

openni::VideoStream Member List

This is the complete list of members for **openni::VideoStream**, including all inherited members.

addNewFrameListener(NewFrameListener *pListener)	**openni::VideoStream**
create(const Device &device, SensorType sensorType)	**openni::VideoStream**
destroy()	**openni::VideoStream**
Device class	**openni::VideoStream**
getCameraSettings()	**openni::VideoStream**
getCropping(int *pOriginX, int *pOriginY, int *pWidth, int *pHeight) const	**openni::VideoStream**
getHorizontalFieldOfView() const	**openni::VideoStream**
getMaxPixelValue() const	**openni::VideoStream**
getMinPixelValue() const	**openni::VideoStream**
getMirroringEnabled() const	**openni::VideoStream**
getProperty(int propertyId, void *data, int *dataSize) const	**openni::VideoStream**
getProperty(int propertyId, T *value) const	**openni::VideoStream**
getSensorInfo() const	**openni::VideoStream**
getVerticalFieldOfView() const	**openni::VideoStream**
getVideoMode() const	**openni::VideoStream**
invoke(int commandId, void *data, int dataSize)	**openni::VideoStream**
invoke(int commandId, T &value)	**openni::VideoStream**
isCommandSupported(int commandId) const	**openni::VideoStream**

isCroppingSupported() const	**openni::VideoStream**
isPropertySupported(int propertyId) const	**openni::VideoStream**
isValid() const	**openni::VideoStream**
readFrame(VideoFrameRef *pFrame)	**openni::VideoStream**
removeNewFrameListener(NewFrameListener *pListener)	**openni::VideoStream**
resetCropping()	**openni::VideoStream**
setCropping(int originX, int originY, int width, int height)	**openni::VideoStream**
setFrameBuffersAllocator(FrameAllocator *pAllocator)	**openni::VideoStream**
setMirroringEnabled(bool isEnabled)	**openni::VideoStream**
setProperty(int propertyId, const void *data, int dataSize)	**openni::VideoStream**
setProperty(int propertyId, const T &value)	**openni::VideoStream**
setVideoMode(const VideoMode &videoMode)	**openni::VideoStream**
start()	**openni::VideoStream**
stop()	**openni::VideoStream**
VideoStream()	**openni::VideoStream**
VideoStream(OniStreamHandle handle)	**openni::VideoStream**
~VideoStream()	**openni::VideoStream**

openni::YUV422DoublePixel Member List

This is the complete list of members for **openni::YUV422DoublePixel**, including all inherited members.

u	openni::YUV422DoublePixel
v	openni::YUV422DoublePixel
y1	openni::YUV422DoublePixel
y2	openni::YUV422DoublePixel

- a -

- addDeviceConnectedListener() : **openni::OpenNI**
- addDeviceDisconnectedListener() : **openni::OpenNI**
- addDeviceStateChangedListener() : **openni::OpenNI**
- addNewFrameListener() : **openni::VideoStream**
- allocateFrameBuffer() : **openni::VideoStream::FrameAllocator**
- Array() : **openni::Array< T >**
- attach() : **openni::Recorder**

- c -

- close() : **openni::Device**
- convertDepthToColor() : **openni::CoordinateConverter**
- convertDepthToWorld() : **openni::CoordinateConverter**
- convertWorldToDepth() : **openni::CoordinateConverter**
- create() : **openni::Recorder** , **openni::VideoStream**

- d -

- destroy() : **openni::VideoStream** , **openni::Recorder**
- Device() : **openni::Device**
- DeviceConnectedListener() : **openni::OpenNI::DeviceConnectedListener**
- DeviceDisconnectedListener() : **openni::OpenNI::DeviceDisconnectedListener**
- DeviceStateChangedListener() :

openni::CameraSettings , **openni::Device** , **openni::Recorder** , **openni::PlaybackControl**

- n -

- NewFrameListener() : **openni::VideoStream::NewFrameListener**

- o -

- onDeviceConnected() : **openni::OpenNI::DeviceConnectedListener**
- onDeviceDisconnected() : **openni::OpenNI::DeviceDisconnectedListener**
- onDeviceStateChanged() : **openni::OpenNI::DeviceStateChangedListener**
- onNewFrame() : **openni::VideoStream::NewFrameListener**
- open() : **openni::Device**
- operator=() : **openni::VideoMode** , **openni::VideoFrameRef**
- operator[]() : **openni::Array< T >**

- r -

- readFrame() : **openni::VideoStream**
- Recorder() : **openni::Recorder**
- release() : **openni::VideoFrameRef**
- removeDeviceConnectedListener() : **openni::OpenNI**
- removeDeviceDisconnectedListener() : **openni::OpenNI**
- removeDeviceStateChangedListener() : **openni::OpenNI**
- removeNewFrameListener() : **openni::VideoStream**
- resetCropping() : **openni::VideoStream**

- s -

- seek() : **openni::PlaybackControl**
- setAutoExposureEnabled() : **openni::CameraSettings**
- setAutoWhiteBalanceEnabled() : **openni::CameraSettings**
- setCropping() : **openni::VideoStream**
- setDepthColorSyncEnabled() : **openni::Device**

- ~DeviceDisconnectedListener() : **openni::OpenNI::DeviceDisconnectedListener**
- ~DeviceStateChangedListener() : **openni::OpenNI::DeviceStateChangedListener**
- ~FrameAllocator() : **openni::VideoStream::FrameAllocator**
- ~NewFrameListener() : **openni::VideoStream::NewFrameListener**
- ~PlaybackControl() : **openni::PlaybackControl**
- ~Recorder() : **openni::Recorder**
- ~VideoFrameRef() : **openni::VideoFrameRef**
- ~VideoStream() : **openni::VideoStream**

OpenNI 2.0

- b : **openni::RGB888Pixel**
- build : **openni::Version**
- g : **openni::RGB888Pixel**
- maintenance : **openni::Version**
- major : **openni::Version**
- minor : **openni::Version**
- r : **openni::RGB888Pixel**
- u : **openni::YUV422DoublePixel**
- v : **openni::YUV422DoublePixel**
- y1 : **openni::YUV422DoublePixel**
- y2 : **openni::YUV422DoublePixel**

- Device : openni::SensorInfo , openni::PlaybackControl , openni::VideoStream , openni::DeviceInfo
- OpenNI : openni::DeviceInfo , openni::OpenNI::DeviceStateChangedListener , openni::OpenNI::DeviceDisconnectedListener , openni::OpenNI::DeviceConnectedListener
- SensorInfo : openni::VideoMode
- VideoFrameRef : openni::VideoMode
- VideoStream : openni::CameraSettings , openni::VideoStream::FrameAllocator , openni::VideoStream::NewFrameListener , openni::VideoFrameRef , openni::SensorInfo , openni::VideoMode

- ANY_DEVICE : **openni**
- TIMEOUT_FOREVER : **openni**
- TIMEOUT_NONE : **openni**

- DepthPixel : **openni**
- Grayscale16Pixel : **openni**

- DeviceState : **openni**
- ImageRegistrationMode : **openni**
- PixelFormat : **openni**
- SensorType : **openni**
- Status : **openni**

- DEVICE_STATE_EOF : **openni**
- DEVICE_STATE_ERROR : **openni**
- DEVICE_STATE_NOT_READY : **openni**
- DEVICE_STATE_OK : **openni**
- IMAGE_REGISTRATION_DEPTH_TO_COLOR : **openni**
- IMAGE_REGISTRATION_OFF : **openni**
- PIXEL_FORMAT_DEPTH_100_UM : **openni**
- PIXEL_FORMAT_DEPTH_1_MM : **openni**
- PIXEL_FORMAT_GRAY16 : **openni**
- PIXEL_FORMAT_GRAY8 : **openni**
- PIXEL_FORMAT_JPEG : **openni**
- PIXEL_FORMAT_RGB888 : **openni**
- PIXEL_FORMAT_SHIFT_9_2 : **openni**
- PIXEL_FORMAT_SHIFT_9_3 : **openni**
- PIXEL_FORMAT_YUV422 : **openni**
- PIXEL_FORMAT_YUYV : **openni**
- SENSOR_COLOR : **openni**
- SENSOR_DEPTH : **openni**
- SENSOR_IR : **openni**
- STATUS_BAD_PARAMETER : **openni**
- STATUS_ERROR : **openni**
- STATUS_NO_DEVICE : **openni**
- STATUS_NOT_IMPLEMENTED : **openni**
- STATUS_NOT_SUPPORTED : **openni**
- STATUS_OK : **openni**
- STATUS_OUT_OF_FLOW : **openni**
- STATUS_TIME_OUT : **openni**

OniEnums.h

Go to the documentation of this file.

```
/*****************************************************************************
*                                                                            *
*  OpenNI 2.x Alpha                                                          *
*  Copyright (C) 2012 PrimeSense Ltd.                                        *
*                                                                            *
*  This file is part of OpenNI.                                              *
*                                                                            *
*  Licensed under the Apache License, Version 2.0 (the "License");           *
*  you may not use this file except in compliance with the License.          *
*  You may obtain a copy of the License at                                   *
*                                                                            *
*      http://www.apache.org/licenses/LICENSE-2.0                            *
*                                                                            *
*  Unless required by applicable law or agre
```

```
ed to in writing, software       *
*  distributed under the License is distribu
ted on an "AS IS" BASIS,         *
*  WITHOUT WARRANTIES OR CONDITIONS OF ANY K
IND, either express or implied.  *
*  See the License for the specific language
 governing permissions and       *
*  limitations under the License.
                                 *
*
                                 *
********************************************
*********************************/
#ifndef _ONI_ENUMS_H_
#define _ONI_ENUMS_H_

namespace openni
{

typedef enum
{
    STATUS_OK = 0,
    STATUS_ERROR = 1,
    STATUS_NOT_IMPLEMENTED = 2,
    STATUS_NOT_SUPPORTED = 3,
    STATUS_BAD_PARAMETER = 4,
    STATUS_OUT_OF_FLOW = 5,
    STATUS_NO_DEVICE = 6,
    STATUS_TIME_OUT = 102,
} Status;

typedef enum
{
    SENSOR_IR = 1,
    SENSOR_COLOR = 2,
    SENSOR_DEPTH = 3,

} SensorType;

typedef enum
{
    // Depth
    PIXEL_FORMAT_DEPTH_1_MM = 100,
    PIXEL_FORMAT_DEPTH_100_UM = 101,
    PIXEL_FORMAT_SHIFT_9_2 = 102,
    PIXEL_FORMAT_SHIFT_9_3 = 103,

    // Color
    PIXEL_FORMAT_RGB888 = 200,
    PIXEL_FORMAT_YUV422 = 201,
    PIXEL_FORMAT_GRAY8 = 202,
    PIXEL_FORMAT_GRAY16 = 203,
    PIXEL_FORMAT_JPEG = 204,
    PIXEL_FORMAT_YUYV = 205,
} PixelFormat;

typedef enum
{
    DEVICE_STATE_OK     = 0,
    DEVICE_STATE_ERROR  = 1,
    DEVICE_STATE_NOT_READY  = 2,
    DEVICE_STATE_EOF    = 3
} DeviceState;

typedef enum
{
    IMAGE_REGISTRATION_OFF              = 0,
    IMAGE_REGISTRATION_DEPTH_TO_COLOR   = 1,
} ImageRegistrationMode;

static const int TIMEOUT_NONE = 0;
static const int TIMEOUT_FOREVER = -1;

} // namespace openni
```

```

#endif // _ONI_ENUMS_H_
```