

OVERVIEW

NDI™ (Network Device Interface) is a standard developed by NewTek to make it easy to prepare products that share video on a local Ethernet network. We believe that the future of the video industry is one in which video is transferred easily and efficiently in IP space, and that this vision will largely supplant current industry-specific connection methods (HDMI, SDI, etc.) in the production pipeline.

In the last prior broadcast industry ‘quantum shift’, cameras and video devices moved from analog to digital formats. In the next, it’s clear that a/v signals are destined to be carried via IP. Already, the vast majority of extant camera and video devices utilize computer-based systems internally and can communicate via IP in some manner. Most video rendering, graphics systems and switchers run on computers too; and many are *already* interconnected over IP. Handling video over the same network opens up a world of new creative and pipeline possibilities.

HOWEVER – THE NDI VISION IS MUCH BROADER AND MORE EXCITING THAN SIMPLY SUBSTITUTING ONE MEANS OF TRANSMISSION BETWEEN DEVICES FOR ANOTHER.

Consider a comparison: The Internet, too, *could* be narrowly described as a transport medium, moving data from point A to point B. Yet, by connecting everyone and everything everywhere together, the ‘net’ is much, much more than the sum of its parts. Likewise, introducing video into the IP realm with its endless potential connections offers exponential creative possibilities along with workflow benefits.

NDI allows multiple video systems to identify and communicate with one another over IP, and to encode, transmit and receive many streams of high quality, low latency, frame-accurate video and audio in real-time. This new protocol can benefit any network-connected video device, including video mixers, graphics systems, capture cards, and many other production devices.

NDI can operate bi-directionally over a local area network, with many video streams on a shared connection. Its encoding algorithm is resolution and frame-rate independent, supporting 4K and beyond, along with 16 channels and more of floating-point audio.

The protocol includes tools that implement video access rights, grouping, bi-directional metadata, and IP commands. Its superb performance over standard GigE networks makes it possible to transition facilities to an incredibly versatile IP video production pipeline without negating existing investments in SDI cameras and infrastructure, or costly new high-speed network infrastructures.

LICENSE

You may use the SDK in accordance with the license that is provided with the SDK, if your platform supports an installer this is provided during the installation process and on all versions this license is available for review from the root of the SDK folder in the file “NDI License Agreement”. Your use of any part of the SDK, for any purpose is acknowledgment that you agree to these license terms.

For distribution, you must implement this within your applications respecting the following requirements:

- Your application must provide a link to <http://NDI.NewTek.com/> in a location that is close to all locations where NDI is used/selected within the product, on your web site, and in its documentation. This will be a landing page that provides all information about NDI and access to the available tools we provide, any updates, and news.
- NDI is a trademark of NewTek and should be used only with the TM as follows: NDI™, along with the statement “NDI™ is a trademark of NewTek, Inc.” located on the same page near the mark where it is first used, or at the bottom of the page in footnotes. Your application’s About Box and any other locations where trademark attribution is provided should also specifically indicate that “NDI™ is a trademark of NewTek, Inc.” For user interfaces, it is also acceptable to use the mark with the NewTek name as “NewTek NDI™”. If you have any questions please do let us know. Note that if you wish to use NDI within the name of your product then you should carefully read the NDI brand guidelines and consult with NewTek.
- You should include the NDI DLLs as part of your own application and keep them in your application folders so that there is no chance that NDI DLLs installed by your application might conflict with other applications on the system that also use NDI. Do not install your NDI DLLs into the system path for this reason.
- A redistributable installer is included with the SDK that you may include, the details of how to locate and use this are described in the section marked “DYNAMIC LOADING OF NDI LIBRARIES”.

We are interested in how our technology is being used and would like to ensure that we have a full list of applications that make use of NDI technology. Please let us know about your commercial application (or interesting non-commercial one) using NDI by emailing ndi@newtek.com.

If you have any questions, comments or requests, please do not hesitate to let us know. Our goal is to provide you with this technology and encourage its use, while at the same time ensuring that both end-users and developers enjoy a consistent high quality experience.

HARDWARE SUPPORT

If you have a need for NDI in embedded or hardware devices then we have a number of options available, including an FPGA design that allows NDI to be compressed in a very small chip (available Q3 2017). Please email ndi@newtek.com if you have an interest in working with NDI outside of the SDK and we will do our absolute best to accommodate you. If you wish to add hardware support for NDI I HX, please contact us.

SOFTWARE DISTRIBUTION

In order to clarify which files may be distributed with your applications the following are the files and the distribution terms under which they may be used. Note that open source projects have the right to include the header files within their distributions; which may then be used with dynamic loading of the NDI libraries.

HEADER FILES. (NDI_SDK_DIR\INCLUDE*.H)

These files may be distributed with open source projects under the MIT license. These headers may be included in open source projects (see “Dynamic Loading” section for preferred mechanism), however the requirements of these projects in terms of visual identification of NDI shall be as outlined within the License section above.

BINARY FILES (NDI_SDK_DIR\BIN*.DLL)

You may distribute these files within your application as long as your EULA terms cover the specific requirements of the NDI SDK EULA and your application covers the terms of the license above. You should place these files within the folder structure of your application to avoid DLL conflicts with other NDI applications on the system. These DLLs require the Visual Studio 2013 C run-times to be installed¹.

REDISTRIBUTABLES (NDI_SDK_DIR\REDIST*.EXE)

You may distribute the NDI redistributables and install them within your own installer; however you should make all reasonable effort to keep the versions that you distribute up to date. You may use the command line with `/verysilent` to install without any user intervention.

An alternative is to provide a user link to the NewTek provided download of this application at <http://new.tk/NDIRedistV3>. At run-time, the location of the NDI run-time DLLs can be determined from the environment variable `NDI_RUNTIME_DIR_V3`.

CONTENT FILES (NDI_SDK_DIR\LOGOS*.*)

You may distribute all files in this folder as you need and use them in any marketing, product or web material.

LIBRARIES

There are three individual libraries that are part of the NDI SDK. These share common structures and conventions to facilitate development, and may all be used together. These are referred to as:

NDI-SEND

This library is used to send video, audio, and meta-data over the network. You establish yourself as a named source on the network and then anyone may see and use the media that you are providing. Video can be sent at any resolution and frame-rate in RGB(+A) and YCbCr color spaces. Any number of receivers can connect to an individual NDI-Send.

NDI-FIND

The finding library is used to locate all of the sources on the local network that are serving media capabilities for use with NDI.

NDI-RECEIVE

The receiving library allows you to take sources on the network and receive them. The SDK internally includes all of the codecs and wraps up all the complexities of reliably receiving high performance network video.

¹ This may be downloaded from:
<https://www.microsoft.com/en-us/download/details.aspx?id=40784>

UTILITIES

NDI includes a number of utilities that can be used for converting between common formats in order to make the library easy to use. For instance, conversion between different audio formats is provided as a service.

CPU REQUIREMENTS

NDI Lib is heavily optimized (much of it is written in assembly); while it detects architecture and uses the best path it can, the minimum required SIMD level is SSSE3 which was introduced by Intel in 2005. Hardware acceleration of streams uses GPU-based fixed function pipelines for decompression to the degree possible; however this is not required, and we will always fall back to software-based compression and decompression.

DYNAMIC LOADING OF NDI LIBRARIES

It is common that you might not want to link directly against the NDI libraries, preferring instead to dynamically load them at run-time (this is of particular value in Open Source projects). There is a structure that contains all of the NDI entry points for a particular SDK version, and you can call a single entry point in the library to recover all of these functions. The basic procedure is relatively simple, and an example provided with the SDK illustrates how this can be performed, however.

LOCATING THE LIBRARY

You can of course include the NDI run-time within your application folder; alternatively, you can install the NDI run-time and use an environment variable to locate it on disk. If you are unable to locate the library on disk, you may ask users to perform a download from a standardized URL. There are system dependent #defines that make this process simple:

`NDILIB_LIBRARY_NAME` is defined to represent the dynamic library name (for example, `Processing.NDI.Lib.x64.dll`).

`NDILIB_REDIST_FOLDER` is an environment variable that references the installed NDI runtime library (for example, `c:\Program Files\NewTek\NDI Redistributable\`).

`NDILIB_REDIST_URL` is a URL where the redistributable for your platform may be downloaded (for example, `http://new.tk/NDIRedistV3`).

RECOVERING THE FUNCTION POINTERS

Once you have located the library, you can look for a single exported function `NDILib_v3_load()`. This function will return a structure of type `NDILib_v3` that gives you a reference to every NDI function.

CALLING NDI FUNCTIONS

Once you have a pointer to `NDILib_v3`, you can replace every function with a simple new reference. For instance, to initialize a sender you can replace a call to `NDILib_find_create_v2` in the following way:

`NDILib_find_create_v2(...)` becomes `p_NDILib->NDILib_find_create_v2(...)`

PERFORMANCE AND IMPLEMENTATION

This section provides some guidelines on how to get the best performance out of the SDK.

UPGRADING YOUR APPLICATIONS

The libraries (dlls) for NDI v3 should be entirely backwards compatible with NDI v2 and you should simply be able to replace these in your application and get most of the benefits of the new version without a single code change.

GENERAL

- Throughout the system, use YCbCr color if possible; it offers both higher performance and better quality.
- If your system has more than one NIC and you are using more than a few senders and receivers, it is worth connecting all available ports to the network. Bandwidth will be distributed across multiple network adapters (requires NDI 2.0 or better).
- Use the latest version of the SDK whenever possible. Naturally, the experience of huge numbers of NDI users in the field provides numerous minor edge-cases, and we work hard to resolve all of these as quickly as possible. As well, we have ambitious plans for the future of NDI and IP video, and we are laying groundwork for these in current versions so that these will already be in place when those features become available for public use.
- The SDK is designed to take advantage of the latest CPU instructions available, particularly AVX2 (256bit instructions) on Intel platforms. Generally, NDI speed limitations relate to system memory bandwidth rather than CPU processing performance. NDI takes advantage of multiple CPU cores when decoding and encoding one or more streams (although the benefits are very small when processing a single stream below HD resolution).
- Please email us at ndi@newtek.com with anything interesting you are doing with the SDK. We are truly interested.
- We take any bugs seriously and are happy to help support you as best we possibly can. If you have any problems please email: ndi@newtek.com

SENDING VIDEO

- Use UYVY or UYVA color if possible, as this avoids internal color conversions. If you cannot generate these color formats and you would use the CPU to perform the conversion, it is better to let the SDK perform the conversion.
- Doing so can yield performance benefits in most cases, particularly when using asynchronous frame submission. If the data that you are sending to NDI is on the GPU and you can have the GPU perform the color conversion before download to system memory, you are likely to find that this has the best performance.
- Sending BGRA or BGRX video will incur a performance penalty. This is caused by the increased memory bandwidth required for these formats, and the conversion into YCbCr color space for compression. With that said, performance has been significantly improved in version 3 of the NDI SDK.
- Using asynchronous frame submission almost always yields significant performance benefits.

RECEIVING VIDEO

- Using `NDIlib_recv_color_format_fastest` for receiving will yield the best performance.
- Having separate threads query for audio and video via `NDIlib_recv_capture` is recommended. Note that `NDIlib_recv_capture` is multi-thread safe, allowing multiple threads to be waiting for data at once. Using a reasonable timeout on `NDIlib_recv_capture` is better and more efficient than polling it with zero time-outs.

- **Experimental.** NDI is very highly optimized, and chooses the best codec possible for use in the most general use cases. Some pipelines through the system include support for hardware accelerated video decoding which can be enabled by sending an XML meta-data message to a receiver as follows :

```
<ndi_hwaccel enabled="true"/>
```

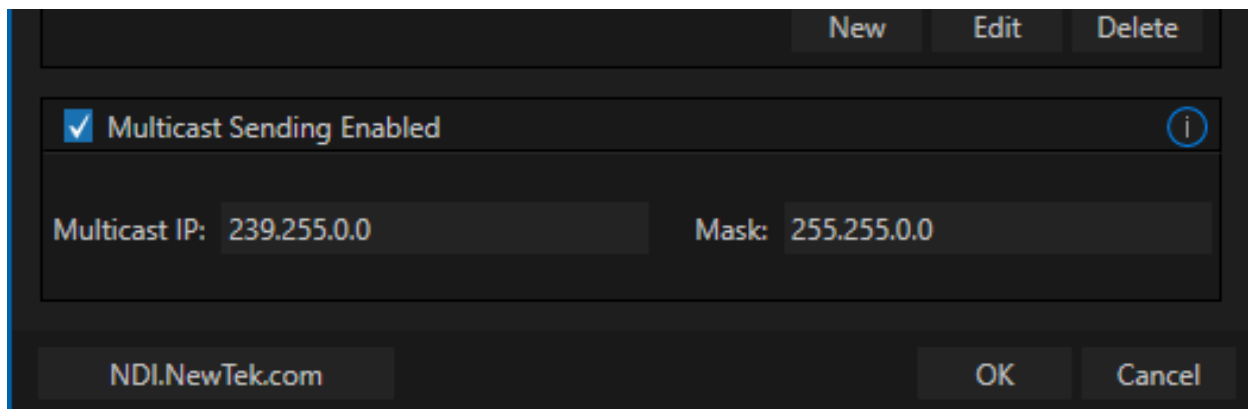
- Bear in mind in that decoding resources on some machines are designed for processing a single video stream. In consequence, while hardware assistance might benefit some small number of streams, it *may* actually hurt performance as the number of streams increases. Thus it is important to realize that there is no “one size fits all” rule respecting hardware acceleration; it can improve performance in some situations, yet degrade performance in others.
- To illustrate: NDI-HX supports a full decode pipeline with hardware acceleration that works on Intel, AMD, nVidia GPUs, and dedicated acceleration devices. On average, hardware acceleration can halve the CPU time taken to decode a frame in many configurations. It is not uncommon, however, that – due to the specifics of the hardware decoding pipeline implementation, which (for example) may use resources that are less well “shared” than the CPU – video decode latency is *higher* with hardware acceleration enabled.

MULTICAST

NDI version 3 includes support for multi-cast. It is important to be aware that using multicast on a network that is not configured correctly is very similar to a “denial of service” attack on the entire network and for this reason the ability to send multicast is disabled by default. The default behavior of every router that we have tested has been to treat multicast traffic as if it was broadcast traffic. Because most multicast traffic on a network is low bandwidth this is of little consequence and in general allows a network router to run more efficiently because no packet filtering is required. What this means is that every multicast packet that is received is sent to every destination on the network regardless of whether it was needed there. Because NDI requires high bandwidth multicast, even with a limited number of sources on a large network the burden of sending this much data to all network sources can cripple the entire network performance.

In order to avoid this problem it is essential to ensure that *every* router on your network has proper multicast filtering enabled. This option is most commonly referred to as “IGMP snooping” which is described in detail at https://en.wikipedia.org/wiki/IGMP_snooping. If you are unable to find a way to enable this option we recommend that use multicast NDI with caution.

If you have not been scared off by the above and have configured your network correctly, you should download the NDI tools and launch the “Access Manager”. This has an option that will enable multicast sending on the current machine. The multicast address range used may also be configured; NDI will automatically detect and choose the multicast address being used without any user configuration and is capable of running even when the available range of addresses is smaller than the number of NDI sources. We do however recommend that you assign the broadest range of IP addresses possible for use. If you specify a MultiCast IP address of “239.255.0.0”, and a mask of “255.255.0.0” then any IP address in the range 239.255.0.0 to 239.255.255.255 will be used.



While you can use the “Access Manager” to enable Multicast with its default settings it is possible on Windows to configure additional settings. All of these settings may be placed in `HKEY_CURRENT_USER` or `HKEY_LOCAL_MACHINE`. If it exists in both then the current user settings are preferred. These different options are outlined below:

SUBNET CONTROL (ON NDI RECEIVER)

By default NDI will multicast to the local subnet and use unicast in order to connect to machines outside of that range; it does this for each network adapter by examining the subnet mask and using that to determine if it can connect with multicast. If you wish to support multicast receiving outside of the local subnet of a sender then on the receiver you should create a registry key of type `REG_MULTI_SZ`, called **SubnetOverrides** in location `[HKCU/HKLM]\Software\NDI\Multicast\Receive`.

This key is a multi-line key that on each line has a subnet in CIDR format that is configured as being accessible to that receiver even if it is not a local subnet. For instance having “10.28.1.100/24” would indicate that IP addresses in the range 10.28.100.0-10.28.100.255 should use multicast. Each line can have an additional IP address range; the default if not present is to only use the local subnet.

TTL CONTROL (ON NDI SENDER)

On a multicast sender you can specify the TTL of the multicast packets, allowing them to traverse multiple network routers by specifying the registry key of type `REG_DWORD`, called **TTL** in the location `[HKCU/HKLM]\Software\NDI\Multicast\Send`. This can take any value in the range 1-255; the default if not present is 1.

DEBUGGING WITH MULTICAST

An important note of caution is that a software application like NDI will subscribe to a multicast group and when it no longer needs that group it will unsubscribe from it. Unlike most operations in the operating system the un-subscription step is not automated by OS which means that once you are subscribed to a group that your computer will continue to receive data until the router sends an IGMP query to verify whether it is still needed which happens about every 5 minutes on typical networks. This means that if you launch an NDI multicast stream and kill your application without closing the NDI connection correctly that your computer will continue to receive the data from the network until this timeout expires.

NDI HIGH EFFICIENCY MODE

NDI-HX is supported by NDI applications starting at SDK v2.1 on Windows and SDK v3.0 on Mac and Linux. A separate NDI-HX driver for the hardware device that is being used must be downloaded (your device documentation will provide you with the link). All HX devices support device recording and web control, with PTZ cameras supporting all PTZ

support. NDI-HX supports hardware assisted compression on many platforms. Drivers are provided on Mac and Windows, if you require support of NDI-HX on Linux within your application please email ndi@newtek.com.

STARTUP AND SHUTDOWN

There are commands `NDILib_initialize()` and `NDILib_destroy()` which can be called to initialize and de-initialize the library. It is recommended (but never required) that you call these. Internally all objects are reference-counted; the libraries are initialized on the first object creation, and they are destroyed on the last, which means that these calls are invoked implicitly.

The only negative side-effect of this behavior is that if you repeatedly create and destroy a single object, more work is done each time than is required. These calls allow that to be avoided. There is no scenario under which these calls can cause a problem, even if you call `NDILib_destroy()` while you still have active objects. `NDILib_initialize()` will return false on an unsupported CPU.

EXAMPLE CODE

The NDI SDK includes a number of examples to help you get going. The following list those examples and what they illustrate.

C# Examples	
Managed NDI Recv	This example illustrates how to use the managed wrapper layer around NDI to find and receive NDI audio and video in a .Net friendly interface.
Managed NDI Router	This example illustrates how to use the managed wrapper layer around NDI to access NDI source routing in a .Net friendly way.
Managed NDI Send	This example illustrates how to use the managed wrapper layer around NDI to send NDI audio and video in a .Net friendly way.
Managed NDILib Send	Illustrates how to use the thin .Net pinvoke wrapper to send NDI audio and video. Very similar to using the C interface.
NDILibDotNet2	Not only an example of .Net pinvoke and use of the NDI library, but also a reusable convenience library for a .Net friendly interface. Used by all .Net examples.
WPF MediaElement Receiver	Illustrates how the DirectShow NDI Source Filter can be used by a WPF Media Element to receive NDI streams.
WPF NDI Send	This example illustrates how to use the NdiSendContainer to send WPF visuals over NDI using only XAML.
C++ Examples	
DShow_Receive_Filter	This illustrates how the DirectShow can be used from C++. You may enter the name of an NDI source on the network and it will use simple graph building to provide an on-screen video window that shows a video source.
NDILib_DynamicLoad	Dynamic loading is the process whereby you do not link directly against the NDI libraries, loading them and connecting to them instead at run-time. This example illustrates how this is done, and is the basis of how one might want to integrate NDI into open source projects distributed under licenses that do not allow the inclusion of external DLLs. This application also illustrates how to take the user to a web site to download the NDI redistributables, if these are not present on the machine.
NDILib_Find	This is a very basic example illustrating how to locate NDI sources on the network. Each time new sources are found or existing sources are removed, it will update the list of sources in a console window.
NDILib_Recv	This is a basic example that illustrates, first, finding the first NDI source on the network, and then connecting to it in order to receive real-time video.
NDILib_Recv_Audio_16bpp	This is very similar to the NDI_recv example, but provided as an example of using functions that operate on 16bpp interleaved audio data.
NDILib_Recv_WebControl	This is a simple example that shows you how to receive an embedded web URL from a device without the need to poll it, and then opens up a web

	browser pointing to that location.
NDIlib_Recv_Record	This shows you how to detect if a source can be recorded. Once it can detect that the source is recordable it will start recording and display the record length.
NDIlib_Recv_PTZ	This shows you how to detect if a source can be PTZ controlled. It then moves the PTZ camera to a particular preset.
NDIlib_Recv_Multichannel	This shows how to connect to a number of NDI sources at once, and receive all of their streams on the local machine.
NDIlib_Routing	Routing is the capability of creating a “virtual NDI source” that then can be pointed at any other NDI sources. This example shows how this may be done.
NDIlib_Send_Audio	This is a simplified example that will create an NDI source and then send it audio data.
NDIlib_Send_Audio_16bpp	This simplified example creates an NDI source and sends it audio data using the 16bpp interleaved audio functions. While lacking the bit precision of the NDI floating point audio support, these functions are often easier to understand.
NDIlib_Send_Benchmark	This creates an image and then passes it into NDI at the highest rate possible. This is meant as a benchmark for local encoding performance. It will provide a video stream on the network that is likely to exceed the available bandwidth and so might drop frames if you connect to it. The purpose of this example is to determine encoder performance on your machine.
NDIlib_Send_BMD	This is an example that will connect to any BlackMagic Design™ cards in your local machine and then present them all as NDI sources so that they can be accessed on your local network at will.
NDIlib_Send_Capabilities	NDI_Capabilities is the mechanism by which NDI senders can provide a user interface to down-stream applications. While it is more common that receivers will write code that receives these messages, this example shows how one might create an NDI sender that provides an interface.
NDIlib_Send_PNG	This is a simple example that loads a PNG with alpha and makes it available as an NDI source.
NDIlib_Send_Video	This is a very simple example that will put an NDI video stream onto the local network.
NDIlib_Send_Video_Advanced	This example illustrates how to send to the network and receive meta-data messages that indicate whether your source is marked as being on program or preview row down-stream.
NDIlib_Send_Video_and_Audio	This illustrates sending audio and video.
NDIlib_Send_Video_Async	In general, NDI’s best performance is achieved by using a separate thread for encoding of video data. This means that your “send” call can return almost immediately, in the assumption that the buffer being sent is not going to be changed until the next frame is ready. This example illustrates this process.
NDIlib_Send_Win32	This is an advanced example showing how you can use some undocumented NDI calls and Win32 to be able to use the standard Win32 processes to generate a real time video output.
VB.Net Examples	
VB NDI Router	This example illustrates how to use the managed wrapper layer around NDI to access NDI source routing in a .Net friendly way.
VB NDI Send	This example illustrates how to use the managed wrapper layer around NDI to send NDI audio and video in a .Net friendly way.
VB NDIlib Recv	Illustrates how to use the thin .Net pinvoke wrapper to receive NDI audio and video. Very similar to using the C interface.
VB NDIlib Send	Illustrates how to use the thin .Net pinvoke wrapper to send NDI audio and video. Very similar to using the C interface.
VB WPF NDI Send	This example illustrates how to use the NdiSendContainer to send WPF visuals over NDI using only XAML.
VB WPF Recv	This example illustrates how to use the managed wrapper layer around NDI to find NDI sources plus receive NDI audio and video.

NDI-SEND

Like all of the NDI libraries, a call to `NDIlib_send_create` will create an instance of the sender, which will return an instance of type `NDIlib_send_instance_t` (or `NULL` if it fails) representing the sending instance. The set of creation parameters that are applied to the sender are specified by filling out a structure called `NDIlib_send_create_t`. It is now possible to call `NDIlib_send_create` with a `NULL` parameter in which case it will use default parameters for all values; it will select the source name by using the current executable name, ensuring that there is a count to ensure sender names are unique (e.g. “My Application”, “My Application 2”, “My Application 3”, etc...)

The parameters supported are as follows:

```
p_ndi_name (const CHAR*)
```

This is the name of the NDI source to create. It is a `NULL`-terminated UTF8 string. This will be the name of the NDI source on the network. For instance, if your network machine name is called “MyMachine” and you specify this parameter as “My Video”, then the NDI source on the network would be “MyMachine (My Video)”.

```
p_groups (const CHAR*)
```

This parameter represents the groups that this NDI sender should place itself into. Groups are sets of NDI sources. Any source can be part of any number of groups, and groups are comma separated. For instance “cameras,studio 1,10am show” would place a source in the three groups named. On the finding side, you can specify which groups to look for, and look in multiple groups. If you specify `NULL` as the groups then the default groups will be used.

The following represents the way in which the group is chosen when the group specified is `NULL`.

1. For “receiving”, the registry path “`HKLM\SOFTWARE\NDI\Groups`” for a value “Receive” can provide a full group string (as described above) that will be used by default. By setting this value, all receiving on your system that has no specific group applied will occur within this defined set of groups.
2. For “sending”, the registry path “`HKLM\SOFTWARE\NDI\Groups`” for a value “Send” can provide a full group string that will be used by default. Much as for receiving, when this string is set and the sources on your system do not specify their own group-set, this will assign all sends on the machine to this defined set of groups.
3. If the registry key is not found, everything is assumed to be within a group called “public”.

```
clock_video, clock_audio (BOOL)
```

These specify whether audio and video “clock” themselves. When they are clocked, video frames added will be rate-limited to match the current framerate that you are submitting at. The same is true for audio. In general if you are submitting video and audio off a single thread then you should only clock one of them (video is probably the better of the two to clock off). If you are submitting audio and video of separate threads then having both clocked can be useful.

A simplified view of the way this works is that when you submit a frame it will keep track of the time that the next frame would be required at. If you then submit a frame before this time, the call will wait until that time. This ensures that, if you sit in a tight loop and render frames as fast as you can go, they will be clocked at the frame-rate that you desire. Note that combining clocked video and audio submission combined with asynchronous frame submission (see below) allows you to write very simple loops to render and submit NDI frames.

An example of creating an NDI sending instance is provided below.

```
NDIlib_send_create_t create_params_Send;  
create_params_Send.p_ndi_name = “My Video”;  
create_params_Send.p_groups = NULL;  
create_params_Send.clock_video = TRUE;  
create_params_Send.clock_audio = TRUE;
```

```
NDIlib_send_instance_t pSend = NDIlib_send_create_v2(&create_params_Send);
if (!pSend) printf("Error creating NDI Sender");
```

Once you have created a device, any NDI finders on the network will be able to see this source as available. You may now send audio, video, or meta-data frames to the device. These may be sent at any time, off any thread, in any order.

There are no reasonable restrictions on video, audio or meta-data frames that can be sent or received. In general, video frames yield better compression ratios as the resolution increases (although the size does increase). Note that all formats can be changed frame-to-frame.

The specific structures used to describe the different frame types are described under the section “Frame types” below. An important note of understanding is that video frames are “buffered” on an input – so that if you provide a video frame to the SDK when there are no current connections to it, when a new incoming connection is received, the last video frame will automatically be sent to it. This is done without any need to recompress a frame (it is buffered in memory in compressed form). The following represents an example of how one might send a single 1080i59.94 white frame over an NDI sending connection.

```
// Allocate a video frame (you would do something smarter than this!)
BYTE* p_frame = (BYTE*)malloc(1920*1080*4);
::memset(p_frame, 255, 1920*1080*4);

// Now send it!
NDIlib_video_frame_v2_t video_frame;
video_frame.xres = 1920;
video_frame.yres = 1080;
video_frame.FourCC = NDIlib_FourCC_type_BGRA;
video_frame.frame_rate_N = 30000;
video_frame.frame_rate_D = 1001;
video_frame.picture_aspect_ratio = 16.0f/9.0f;
video_frame.is_progressive = NDIlib_frame_format_type_progressive;
video_frame.timecode = 0LL;
video_frame.p_data = p_frame;
video_frame.line_stride_in_bytes = 1920*4;
video_frame.p_metadata = "<Hello/>";

// Submit the buffer
NDIlib_send_send_video_v2(pSend, &video_frame);

// Free video memory
free(p_frame);
//In a similar fashion, audio can be submitted for NDI audio sending, //the following will submit
1920 quad-channel silent audio samples at //48kHz
Allocate an audio frame (you would do something smarter than this!);
float* p_frame = (float*)malloc(sizeof(float)*1920*4)
::memset(p_frame, 0, sizeof(float)*1920*4);

// Describe the buffer
NDIlib_audio_frame_v2_t audio_frame;
audio_frame.sample_rate = 48000;
audio_frame.no_channels = 4;
audio_frame.no_samples = 1920;
audio_frame.timecode = 0LL;
audio_frame.p_data = p_frame;
audio_frame.channel_stride_in_bytes = sizeof(float)*1920;
audio_frame.p_metadata = NULL; // No meta-data on this example!

// Submit the buffer
NDIlib_send_send_audio_v2(pSend, &audio_frame);
```

```
// Free the audio memory
free(p_frame);
```

Because many applications like providing interleaved 16bpp audio, the NDI library includes utility functions that will convert in and out of floating point formats from PCM 16bpp formats. Alternatively, there is a utility function (`NDIlib_util_send_send_audio_interleaved_16s`) for sending signed 16 bit audio using. We would refer you to the example projects and also the header file `Processing.NDI.utilities.h` which lists the functions available. In general we recommend that you use floating point audio since clamping is not possible and audio levels are well defined without a need to consider audio headroom.

Metadata is submitted in a very similar fashion. (We do not provide a code example, since it is easily understood by referring to the audio and video examples.)

In order to receive metadata being sent from the receiving end of a connection (e.g. which can be used to select pages, change settings, etc.) we would refer you to the way in which the receive device works. The basic process involves calling `NDIlib_send_capture` with a time-out value. This can be used either to query whether there is a metadata message available if the time-out is zero, or can be used on a thread to efficiently wait for messages. The basic process is outlined below:

```
// Wait for 1 second to see if there is a metadata message available
NDIlib_metadata_frame_t meta_data;
if (NDIlib_send_capture(pSend, &meta_data, 1000)==NDIlib_frame_type_metadata)
{   // Do something with the meta-data here
    // ...

    // Free the meta data message
    NDIlib_rcv_free_metadata(pSend, &meta_data);
}
```

An important category of meta-data that you will receive automatically when new connections to you are established is connection meta-data, as specified in the NDI-Recv section of this documentation. This allows an NDI receiver to provide up-stream details to a sender that might indicate hints as to what capabilities that the receiver might have. Examples include what resolution and frame-rate is preferred by the receiver, what its product name is, etc.

It is important that a sender is aware that it might be sending video data to more than one receiver at a time, and in consequence will receive connection meta-data from each one of them.

Determining whether you are on program and/or preview output on a device such as a video mixer (i.e., ‘Tally’ information) is very similar to how you handle metadata information. You can ‘query’ it, or you can efficiently ‘wait’ and get tally notification changes. The following example will wait for one second and react to tally notifications:

```
// Wait for 1 second to see if there is a tally change notification.
NDIlib_tally_t tally_data;
if (NDIlib_send_get_tally(pSend, &tally_data)==TRUE)
{   // The tally state changed and you can now
    // read the new state from tally_data.
}
```

An NDI send instance is destroyed by passing it into `NDIlib_send_destroy`.

Connection metadata is data that you can “register” with a sender and it will automatically be sent each time a new connection with the sender is established. The sender will internally maintain a keep a copy of any connection metadata messages that you have and send them automatically.

This is useful to allow a sender to provide downstream information at the time of connection to any device that might want to connect to it (for instance, letting it know what the product name or preferred video format might be). Neither senders nor receivers are required to provide this functionality, and may freely ignore any connection data strings.

Standard connection metadata strings are defined in a later section of this document. In order to add a meta-data element, one can call `NDIlib_send_add_connection_metadata`; to clear all of the registered elements, one can call `NDIlib_send_clear_connection_metadata`.

An example that registers the name and details of your sender so that other sources that connect to you get information about what you are is provided below.

```
// Provide a meta-data registration that allows people to know what we are.
static const char* p_connection_string =
```

```
NDIlib_metadata_frame_t NDI_connection_type;
NDI_connection_type.p_data =
"<ndi_product long_name=\"NDIlib Send Example.\" \" \"
  \"          short_name=\"NDIlib Send\" \" \"
  \"          manufacturer=\"CoolCo, inc.\" \" \"
  \"          model_name=\"PBX-15M\" \" \"
  \"          version=\"1.000.000\" \" \"
  \"          serial=\"ABCDEFGF\" \" \"
  \"          session_name=\"My Midday Show\"/>\";

NDIlib_send_add_connection_metadata(pNDI_send, &NDI_connection_type);
```

ASYNCHRONOUS SENDING

It is possible to send video frames asynchronously using NDI, using the call `NDIlib_send_send_video_v2_async`. This function will return immediately, and will perform all required operations (including color conversion, any compression and network transmission) asynchronously with the call.

Because NDI takes full advantage of asynchronous OS behavior when available, this will normally result in improved performance (as compared to creating your own thread and submitting frames asynchronously with rendering). The memory that you passed to the API through the `NDIlib_video_frame_v2_t` pointer will continue to be used until a synchronizing API call is made.

Synchronizing calls are any of the following:

- Another call to `NDIlib_send_send_video_v2_async`.
- A call to `NDIlib_send_send_video_v2_async(pNDI_send, NULL)` will wait for any asynchronously scheduled frames to completed and then return. Obviously you can also submit the next frame, whereupon it will wait for the previous frame to finish before asynchronously submitting the current one.
- Another call to `NDIlib_send_send_video_v2`.
- A call to `NDIlib_send_destroy`.

Using this in conjunction with a clocked video output results in a very efficient rendering loop where you do not need to use separate threads for timing or for frame submission. In other words, the following is an efficient real-time processing system as long as rendering can always keep up with real-time.

```
while(!done())
{
    render_frame();
    NDIlib_send_send_video_v2_async(pNDISend, &frame_data);
}
NDIlib_send_send_video_v2_async(pNDISend, NULL); // Sync here
```

TIMECODE SYNTHESIS

It is possible to specify your own timecode for all data sent when sending video, audio or metadata frames. You may also specify a value of `NDIlib_send_timecode_synthesize` (defined as `INT64_MAX`) that will instruct the SDK to generate the timecode for you. When you specify this as a timecode, the timecode will be synthesized for you as UTC time since the Unix Epoch (1/1/1970 00:00) with 100nS precision,

If you never specify a timecode at all, and instead ask for each to be synthesized, then this will use the current system clock time as the starting timecode (translated to UTC since the Unix Epoch) and generate synthetic values, keeping your streams exactly in sync (as long as the frames you are sending do not deviate from the system time in any meaningful way). In practice this means that if you never specify timecodes, they will always be generated correctly for you.

Timecodes coming from different senders on the same machine will always be in sync with each other when working in this way. If you have NTP installed on your local network, then streams can be synchronized between multiple machines with very high precision.

If you specify a timecode at a particular frame (audio or video), then ask for all subsequent ones to be synthesized. The subsequent ones will be generated to continue this sequence, maintaining the correct relationship both the between streams and samples generated, avoiding their deviating over time in any meaningful way from the timecode that you specified.

If you specify timecodes on one stream (e.g. video) and ask for the other stream (audio) to be synthesized, the correct timecodes will be generated for the other stream and will be synthesized exactly to match (they are not quantized inter-streams) the correct sample positions. This ensures that you can specify just the timecodes on a single stream and have the system generate the others for you.

When you send metadata messages and ask for the timecode to be synthesized, it is chosen to match the closest audio or video frame timecode so that it looks close to something you might want - unless there is no sample that looks close; in the latter case, a timecode is synthesized from the last ones known and the time elapsed since it was sent.

Note that the algorithm to generate timecodes synthetically will correctly assign timestamps if frames are not submitted at the exact time.

For instance, if you submit a video frame and then an audio frame in sequential order, they will both correctly have the same timecode – even though it is possible that the video frame took a few milliseconds to encode. That said, no per-frame error is ever accumulated; so, if you are submitting audio and video and they do not align over a period of more than a few frames, the timecodes will still be correctly synthesized without accumulated error.

FAILSAFE

Failsafe is a capability of any NDI sender. The basic capability is that - if you specify a failsafe source on an NDI sender - if the sender were to fail for any reason (even the machine failing completely), any receivers who are viewing that sender will automatically switch over to the failsafe sender. If the failed source comes back online in the meantime, receivers will switch back to that source.

You can set the fail-over source on any video input with a call to:

```
void NDIlib_send_set_failover(NDIlib_send_instance_t p_instance,
                             const NDIlib_source_t* p_failover_source);
```

The failover source can be any network source. If it is specified as `NULL`, the failsafe source will be cleared.

APPLE IOS NOTES

When an iOS app is sent to the background, most of the networking functionality is put into a suspended state. Sometimes resources associated with networking are released back to the operating system while in this state. Apple recommends that certain networking operations be closed down when the app is placed in to the background, then restarted upon being put into the foreground again. Because of this recommendation, we recommend releasing an NDI sender instance within the app's `applicationDidEnterBackground` method, then recreating the instance in the `applicationDidBecomeActive` method.

NDI-FIND

This SDK is provided to locate sources available on the network, and is normally used in conjunction with the NDI-Receive SDK. Internally, it uses a cross-process P2P mDNS implementation to locate sources on the network. It commonly takes a few seconds to locate all of the sources available, since this requires other running machines to send response messages.

Although discovery uses mDNS, the client is entirely self-contained; Bonjour (etc.) are not required. mDNS is a P2P system that exchanges located network sources, and provides a highly robust and bandwidth-efficient way to perform discovery on a local network. On mDNS initialization (often done using the NDI-Find SDK), a few seconds might elapse before all sources on the network are located.

Some network routers might block mDNS traffic between network segments.

Creating the find instance is very similar to the other APIs. One fills out a `NDIlib_find_create_t` structure to describe the device that is needed. It is possible to specify a NULL creation parameter in which case default parameters are used, if you wish to specify the parameters manually, then the member values are as follows:

`show_local_sources` (BOOL)

This flag will tell the finder whether it should locate and report NDI send sources that are running on the current local machine.

`p_groups` (const CHAR*)

This parameter specifies groups for which this NDI finder will report sources. A full description of this parameter and what a NULL default value means is provided in the description of the NDI-Send SDK.

`p_extra_ips` (const CHAR*)

This parameter will specify a comma separated list of IP addresses that will be queried for NDI sources and added to the list reported by NDI find. These IP addresses need not be on the local network, and can be in any IP visible range. NDI find will be able to find and report any number of NDI sources running on remote machines, and will correctly observe them coming online and going offline.

Once you have a handle to the NDI find instance, you can recover the list of current sources by calling `NDIlib_find_get_current_sources` at any time. This will *immediately* return with the current list of located sources. The pointer returned by `NDIlib_find_get_current_sources` is owned by the finder instance, so there is no reason to free it. It will be retained until the next call to `NDIlib_find_get_current_sources`, or until the `NDIlib_find_destroy` function is destroyed.

In order to wait until the set of network sources has been changed, you can call `NDIlib_find_wait_for_sources`. This takes a time-out in milliseconds. If a new source is found on the network or one has been removed before this time has

elapsed, the function will return true immediately. If no new sources are seen before the time has elapsed it will return false.

The following code will create an NDI-Find instance, and then list the current available sources. It uses `NDIlib_find_wait_for_sources` to sleep until new sources are found on the network and, when they are seen, it will call `NDIlib_find_get_current_sources` to get the current list of sources.

```
// Create the descriptor of the object to create
NDIlib_find_create_t find_create;
find_create.show_local_sources = TRUE;
find_create.p_groups = NULL;

// Create the instance
NDIlib_find_instance_t pFind = NDIlib_find_create_v2(&find_create);
if (!pFind) /* Error */;

while(true) // You would not loop forever of course !
{ // Wait up till 5 seconds to check for new sources to be added or removed
    if (!NDIlib_find_wait_for_sources(pNDI_find, 5000))
    { // No new sources added !
        printf("No change to the sources found.\n");
    }
    else
    { // Get the updated list of sources
        uint32_t no_sources = 0;
        const NDIlib_source_t* p_sources = NDIlib_find_get_current_sources(pNDI_find,
&no_sources);

        // Display all the sources.
        printf("Network sources (%u found).\n", no_sources);
        for (uint32_t i = 0; i < no_sources; i++)
            printf("%u. %s\n", i + 1, p_sources[i].p_ndi_name);
    }
}

// Destroy the finder when you're all done finding things
NDIlib_find_destroy(pFind);
```

It is important to understand that mDNS discovery might take some time to locate all network sources. This means that an ‘early’ return to `NDIlib_find_get_current_sources` might not include all of the sources on the network; these will be added (or removed) as additional or new sources are discovered. It is common that it takes a few seconds to discover all sources on a network.

For applications that wish to list the current sources in a user interface menu, the recommended approach would be to create an `NDIlib_find_instance_t` instance when your user interface is opened and then – each time you wish to display the current list of available sources – you can call `NDIlib_find_get_current_sources`.

NDI-RECV

The NDI receive SDK is how frames are received over the network. It is important to be aware that it can connect to sources and remain “connected” to them even when they are no longer available on the network; it will automatically reconnect if the source becomes available again.

As with the other APIs, the starting point is to use the `NDIlib_recv_create_v3` function. This function may be initialized with NULL and default settings are used. This takes settings defined by `NDIlib_recv_create_v3_t`, as follows below:

source_to_connect_to

This is the source name that should be connected too. This is in the exact format returned by

`NDIlib_find_get_sources`. Note that you may specify the source as a NULL source if you wish to create a receiver that you desire to connect at a later point with `NDIlib_recv_connect`.

p_ndi_name

This is a name that is used for the receiver and will be used in future versions of the SDK to allow discovery of both senders and receivers on the network. This can be specified as NULL and a unique name based on the application executable name will be used.

color_format

This parameter determines what color formats you are passed when a frame is received. In general, there are two color formats used in any scenario: that which exists when the source has an alpha channel, and that when it does not. The following represent the formats that are used to

Value	Color format for frames with no alpha channel	Color format for frames with alpha channel
<code>NDIlib_recv_color_format_BGRX_BGRA</code>	BGRX	BGRA
<code>NDIlib_recv_color_format_UYVY_BGRA</code>	UYVY	BGRA
<code>NDIlib_recv_color_format_RGBX_RGBA</code>	RGBX	RGBA
<code>NDIlib_recv_color_format_UYVY_RGBA</code>	UYVY	RGBA
<code>NDIlib_recv_color_format_fastest</code>	Normally UYVY See notes below.	Normally UYVA See notes below.

If you specify the color option `NDIlib_recv_color_format_fastest`, the SDK will provide you buffers in the format that it processes internally, without performing any conversions before they are passed to you. This results in the best possible performance. This option also typically runs with lower latency than other options, since it supports “single field” format types. The `allow_video_fields` option is assumed to be true when in this mode.

You should support the `NDIlib_video_frame_v2_t` properties as widely as you possibly can in this mode, since there are very few restrictions on what you might be passed.

bandwidth

This allows you to specify whether this connection is in high or low bandwidth mode. It is an enumeration, because it is possible that other alternatives will be available in the future. For most uses you should specify

`NDIlib_recv_bandwidth_highest` which will result in the same stream that is being sent from the up-stream source to you.

You may specify `NDIlib_recv_bandwidth_lowest` which will provide you with a medium quality stream that takes almost significantly reduced bandwidth.

allow_video_fields

If your application does not like receiving fielded video data then you can specify `FALSE` to this value and all video you receive will be de-interlaced before it is passed to you. The default value should be considered `TRUE` for most applications. This flag has an implied value of `TRUE` when `color_format` is `NDIlib_recv_color_format_fastest`.

p_ndi_name

This is the name of the NDI receiver to create. It is a NULL-terminated UTF8 string. Give your receiver a meaningful, descriptive, and unique name. This will be the name of the NDI receiver on the network. For instance, if your network machine name is called “MyMachine” and you specify this parameter as “Video Viewer”, then the NDI receiver on the network would be “MyMachine (Video Viewer)”.

Once you have filled out this structure, calling `NDIlib_recv_create_v3` will create an instance for you. A full example is provided with the SDK that illustrates finding a network source and creating a receiver to view it (we will not reproduce that code here).

If you create a receiver with NULL as the settings, or if you wish to change the remote source that you are connected to then you may call `NDIlib_recv_connect` at any time with a `NDIlib_source_t` pointer. If the source pointer is NULL then it will disconnect you from any sources to which you are connected.

Once you have a receiving instance, you can query it for video, audio, or meta-data frames by calling `NDIlib_recv_capture`. This function takes a pointer to the header for audio (`NDIlib_audio_frame_v2_t`), video (`NDIlib_video_frame_v2_t`) and metadata (`NDIlib_metadata_frame_t`), any of which can be NULL. This function can safely be called across many threads at the same time, allowing you to easily have one thread receiving video while another receives audio.

The function takes a timeout value specified in milliseconds. If you call `NDIlib_recv_capture` and there is a frame available then it will be returned without any internal waiting or locking of any kind. If the timeout is zero, it will return immediately with a frame if there is one. If the timeout is not zero, it will wait for a frame up to the timeout duration specified, and return if it gets one (if there is already a frame waiting when the call is made it will return that frame immediately). If a frame of the type requested has been received before the timeout occurs, the function will return the data type received. Frames returned to you by this function must be freed.

The following code illustrates how one might receive audio and/or video based on what is available; it will wait one second before returning if no data was received;

```
NDIlib_video_frame_v2_t video_frame;
NDIlib_audio_frame_v2_t audio_frame;
NDIlib_metadata_frame_t metadata_frame;

switch(NDIlib_recv_capture_v2(pRecv, &video_frame, &audio_frame, &metadata_frame, 1000 ))
{
    // We received video.
    case NDIlib_frame_type_video:
        // Process video here
        // Free the video.
        NDIlib_recv_free_video_v2(pRecv, &video_frame);
        break;

    // We received audio.
    case NDIlib_frame_type_audio:
        // Process audio here
        // Free the audio.
        NDIlib_recv_free_audio_v2(pRecv, &audio_frame);
        break;

    // We received a meta-data packet
    case NDIlib_frame_type_metadata:
        // Do what you want with the meta-data message here.
        // Free the message
        NDIlib_recv_free_metadata(pRecv, &metadata_frame);
        Break;

    // No audio or video has been received in the time-period.
    case NDIlib_frame_type_none:
        break;

    // The device has changed status in some way (see notes below)
    case NDIlib_frame_type_status_change:
        break;
}
```

You are able, if you wish, to take the received video, audio, or metadata frames and free them on another thread to ensure that there is no chance of dropping frames while receiving them. A short queue is maintained on the receiver to allow you to process incoming data in the fashion that is the most convenient within your application. If you always

process buffers faster than real-time this queue will always be empty, and you will be running at the lowest possible latency.

When the value `NDIlib_frame_type_status_change` is returned from `NDIlib_recv_capture_v2` or `NDIlib_recv_capture`, this indicates that the properties of the device have changed. Because connecting to a video source might take a few seconds some of the properties of that device are not known immediately and might even change on the fly. For instance, when connecting to a PTZ camera it might not be known for a few seconds that it supports the PTZ command set. When it becomes known that it supports PTZ the value `NDIlib_frame_type_status_change` is returned indicating that one should recheck device properties. This setting is currently sent when a source changes PTZ type, recording capabilities or web user interface control.

If you wish to determine whether any audio, video or meta-data frames have been dropped, you can call `NDIlib_recv_get_performance`, which will tell you the total frame counts and also the number of frames that have been dropped because they could not be de-queued fast enough.

If you wish to determine the current queue depths on audio, video or meta-data (in order to poll whether receiving a frame would immediately give a result), you can call `NDIlib_recv_get_queue`.

`NDIlib_recv_get_no_connections` will return the number of connections that are currently active, and can also be used to detect whether the video source you are connected to is currently online or not.

Additional function provided by the receive SDK allows metadata to be passed upstream to connected sources via `NDIlib_recv_send_metadata`. Much like the sending of metadata frames in the NDI Send SDK, this is passed as an `NDIlib_metadata_frame_t` structure that is to be sent.

Tally information is handled via `NDIlib_recv_set_tally`. This will take a `NDIlib_tally_t` structure that can be used to define the program and preview visibility status. The tally status is retained within the receiver so that, even if a connection is lost, the tally state is correctly set when it is subsequently restored.

Connection meta-data is an important concept that allows you to “register” certain meta-data messages so that – each time a new connection is established – the up-stream source (normally an NDI Send user) would receive those strings. Note that there are many reasons that connections might be lost and established at run-time. For instance, if an NDI-Sender went offline then the connection is lost; if it comes back online at a later time, the connection would be re-established and the connection meta-data would be resent.

Some standard connection strings are specified for connection metadata, as outlined in the next section. Connection meta-data strings are added with `NDIlib_recv_add_connection_metadata` that takes an `NDIlib_metadata_frame_t` structure. To clear all connection metadata strings allowing them to be replaced, call `NDIlib_recv_clear_connection_metadata`. An example that illustrates how you can provide your product name to anyone who ever connects to you is provided below.

```
// Provide a meta-data registration that allows people to know what we are.
NDIlib_metadata_frame_t NDI_connection_type;
NDI_connection_type.p_data =
    "<ndi_product long_name=\"NDIlib Receive Example.\" "
    "          short_name=\"NDIlib Receive\" "
    "          manufacturer=\"CoolCo, inc.\" "
    "          version=\"1.000.000\" "
    "          model_name=\"PBX-42Q\" "
    "          session_name=\"My Midday Show\" "
    "          serial=\"ABCDEFGH\"/>";

NDIlib_recv_add_connection_metadata(pNDI_recv, &NDI_connection_type);
```

RECEIVER USER INTERFACES

A sender might provide an interface that allows configuration. For instance a NDI converter device might offer an interface that allows it's settings to be change or a PTZ camera might provide an interface that provides access to specific setting and mode values. These interfaces are provided via a web URL that you can host. For instance a converter device might have an embedded web page that is served at a URL such as <http://192.168.1.156/control/index.html> . In order to get this address you simply call the function:

```
const char* NDILib_rcv_get_web_control(NDILib_rcv_instance_t p_instance);
```

This will return a string that represents the URL or NULL if there is no current URL associated with the sender in question. Because connections might take a few seconds this string might not be available immediately after having called connect. To avoid the need to poll this setting, you can know when this setting is now known, or when it has changed by when `NDILib_rcv_capture_v2` or `NDILib_rcv_capture` return a value of `NDILib_frame_type_status_change`.

The string returned is owned by your application until you call `NDILib_rcv_free_string`. An example to recover this is illustrated below:

```
const char* p_url = NDILib_rcv_get_web_control(p_NDIrcv);
if (p_url)
{
    // You now have a URL that you can embed in your user interface if you want!
    // Do what you want with it here ... and when done, call:
    NDILib_rcv_free_string(p_NDIrcv, p_url);
}
else
{
    // This device does not currently support a configuration user interface.
}
```

You can then store this URL, and provide it to an end user as the options for that device. For instance, a PTZ camera or an NDI conversion box might allow its settings to be configured using a hosted web interface. NewTek's Studio Monitor application includes this capability for sources indicating the ability to be configured, as shown in the bottom-right corner of the image below.



When you click this gear gadget, the application will open the web page specified by the sender.

RECEIVER PTZ CONTROL



NDI standardizes the control of PTZ cameras. An NDI receiver will automatically sense whether the device that it is connected too is a PTZ camera and whether it may be controlled automatically.

When controlling a camera via NDI, all configuration of the camera is completely transparent to the NDI client, which will respond to a uniform set of standard commands with well-defined

parameter ranges. For instance, NewTek's Studio Monitor application uses these commands to display PTZ on-screen controls when the source being viewed is reported to be a camera that can be controlled.

In order to determine whether the connection that you are on would respond to PTZ messages you may simply ask the receiver whether this property is supported by calling:

```
bool NDilib_recv_ptz_is_supported(NDilib_recv_instance_t p_instance);
```

This will return true when the video source is a PTZ system and false otherwise. Note that because connections are not instantaneous you might need to wait a few seconds after connection in order for the source to know whether it supports PTZ control. To avoid the need to poll this setting, you can know when this setting is now known, or when it has changed by when `NDilib_recv_capture_v2` or `NDilib_recv_capture` return a value of `NDilib_frame_type_status_change`.

PTZ CONTROL

There are standard API functions that will execute the standard set of PTZ commands. This list is not designed to be exhaustive and might be expanded in the future, it is generally recommended that PTZ cameras provide a web interface to give access to the full set of capabilities of the camera and the host application control the basic messages below.

ZOOM LEVEL

```
bool NDilib_recv_ptz_zoom(NDilib_recv_instance_t p_instance, const float zoom_value);
```

Set the camera zoom level. The zoom value ranges from 0.0 to 1.0.

```
bool NDilib_recv_ptz_zoom_speed(NDilib_recv_instance_t p_instance, const float zoom_speed);
```

Control the zoom level as a speed value. The zoom speed value is in the range [-1.0, +1.0] with zero indicating no motion.

PAN AND TILT

```
bool NDilib_recv_ptz_pan_tilt_speed(NDilib_recv_instance_t p_instance, const float pan_speed,
                                     const float tilt_speed);
```

This will tell the camera to move with a specific speed toward a direction. The speed is specified in a range [-1.0, 1.0], with 0.0 meaning no motion.

```
bool NDilib_recv_ptz_pan_tilt(NDilib_recv_instance_t p_instance, const float pan_value,
                              const float tilt_value);
```

This will set the absolute values for pan and tilt. The range of these values is [-1.0, +1.0] with 0.0 representing center.

PRESETS

```
bool NDilib_recv_ptz_store_preset(NDilib_recv_instance_t p_instance, const int preset_no);
```

Store the current camera position as a preset. The preset number is in the range 0 to 99.

```
bool NDilib_recv_ptz_recall_preset(NDilib_recv_instance_t p_instance, const int preset_no,
                                   const float speed);
```

Recall a PTZ preset. The preset number is in the range 0 to 99. The speed value is in the range 0.0 to 1.0, and controls how fast it will move to the preset,

FOCUS

Focus on cameras can either be in auto-focus mode or in manual focus mode. The following commands are examples of these commands:

```
bool NDlib_recv_ptz_auto_focus(NDlib_recv_instance_t p_instance);  
bool NDlib_recv_ptz_focus(NDlib_recv_instance_t p_instance, const float focus_value);
```

If the mode is auto, then there are no other settings. If the mode is manual, then the value is the focus distance, specified in the range 0.0 to 1.0.

If you wish to control the focus by speed instead of absolute value, you may do this as follows:

```
bool NDlib_recv_ptz_focus_speed(NDlib_recv_instance_t p_instance, const float focus_speed);
```

The focus speed is in the range -1.0 to +1.0, with 0.0 indicating no change in focus value.

WHITE BALANCE

White balance can be in a variety of modes, including the following examples

```
bool NDlib_recv_ptz_white_balance_auto(NDlib_recv_instance_t p_instance);
```

This will place the camera in auto-white balance mode.

```
bool NDlib_recv_ptz_white_balance_indoor(NDlib_recv_instance_t p_instance);
```

This will place the camera in auto-white balance mode, but with a preference for indoor settings.

```
bool NDlib_recv_ptz_white_balance_outdoor(NDlib_recv_instance_t p_instance);
```

This will place the camera in auto-white balance mode, but with a preference for outdoor settings.

```
bool NDlib_recv_ptz_white_balance_manual(NDlib_recv_instance_t p_instance,  
                                         const float red, const float blue);
```

This allows for manual white-balancing, with the red and blue values in the range 0.0 to 1.0.

```
bool NDlib_recv_ptz_white_balance_oneshot(NDlib_recv_instance_t p_instance);
```

This allows you to setup the white-balance automatically using the current center of the camera position. It will then store that value as the white-balance setting.

EXPOSURE CONTROL

Exposure can either be automatic or manual.

```
bool NDlib_recv_ptz_exposure_auto(NDlib_recv_instance_t p_instance);
```

This will place the camera in auto exposure mode.

```
bool NDlib_recv_ptz_exposure_manual(NDlib_recv_instance_t p_instance,  
                                     const float exposure_level);
```

This will place the camera in manual exposure mode with a value in the range [0.0, 1.0].

REMOTE VIDEO RECORDING

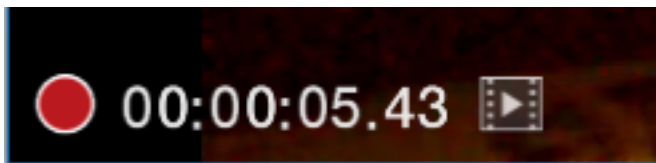
NDI includes a standard way of initiating and accessing remote recordings. The benefit of this is that, when you have a large number of video sources available, you can in effect control them all and record as many of them as needed. For instance all NDI HX devices support recording into an MP4 files, allowing any reasonable number to be recorded on a system; thus a full ISO recording workflow in which every source can be individually controlled and has minimal overhead on the receiving system.

To determine whether a particular NDI source is able to record, you may call the function, :

```
bool NDILib_recv_recording_is_supported(NDILib_recv_instance_t p_instance);
```

This will return true when the video source is able to record and false otherwise. Note that because connections are not instantaneous you might need to wait a few seconds after connection in order for the source to know whether it supports PTZ control. To avoid the need to poll this setting, you can know when this setting is now known, or when it has changed by when `NDILib_recv_capture_v2` or `NDILib_recv_capture` return a value of `NDILib_frame_type_status_change`.

NewTek's Studio Monitor application uses this feature to determine if a network source can support recording and displays record controls as required based on this.



When a message like this is received, it indicates that the source is able to record.

CONTROLLING RECORDING

Record commands that may be sent to a remote recording device are listed below.

START RECORDING

```
bool NDILib_recv_recording_start(NDILib_recv_instance_t p_instance, const char* p_filename_hint);
```

This will start recording. If the recorder was already recording then the message is ignored. A filename is passed in as a 'hint'. Since the recorder might already be recording (or might not allow complete flexibility over its filename), the filename might or might not be used. If the filename is empty, or `NULL`, a name will be chosen automatically.

STOP RECORDING

```
bool NDILib_recv_recording_stop(NDILib_recv_instance_t p_instance);
```

This will stop recording.

SET AUDIO LEVEL

```
bool NDILib_recv_recording_set_audio_level(NDILib_recv_instance_t p_instance,
                                           const float level_dB);
```

This will control the audio level for the recording. dB is specified in decibels relative to the reference level of the source. Not all recording sources support controlling audio levels. For instance, a digital audio device would not be able to avoid clipping on sources already at the wrong level, thus might not support this message.

RECORDING FEEDBACK

Recording will provide feedback on the current recording status.

PROGRESS

```
bool NDILib_rcv_recording_is_recording(NDILib_rcv_instance_t p_instance);
```

This will inform you on whether if the source is currently recording. It will return true while recording is in progress and false when it is not. Because there is one recorded and multiple people might be connected to it, there is a chance that it is recording which was initiated by someone else.

FILENAMES

```
const char* NDILib_rcv_recording_get_filename(NDILib_rcv_instance_t p_instance);
```

When you initiate recording, this is set as set soon as the file is being recorded. Get the current filename for recording. When this is set it will return a non-NULL value which is owned by you and freed using `NDILib_rcv_free_string`. If a file was already being recorded by another client, the message will contain the name of that file. The filename contains a UNC path (when one is available) to the recorded file, and can be used to access the file on your local machine for playback. This will remain valid even after the file has stopped being recorded until the next file is started.

ERROR REPORTING

This will tell you whether there was a recording error and what that string is. When this is set it will return a non-NULL value which is owned by you and freed using `NDILib_rcv_free_string`. When there is no error it will return NULL.

```
const char* NDILib_rcv_recording_get_error(NDILib_rcv_instance_t p_instance);
```

FILE TIMING INFORMATION

There is a structure that gives you information about the current recording.

```
typedef struct NDILib_rcv_recording_time_t
{
    // The number of actual video frames recorded.
    int64_t no_frames;

    // The starting time and current largest time of the record, in UTC time, at 100ns unit
    // intervals. This allows you to know the record time irrespective of frame-rate. For
    // instance, last_time - start_time would give you the recording length in 100ns
    // intervals.
    int64_t start_time, last_time;
} NDILib_rcv_recording_time_t;
```

In order to get this you would call the function :

```
const bool NDILib_rcv_recording_get_times(NDILib_rcv_instance_t p_instance,
                                           NDILib_rcv_recording_time_t* p_times);
```

RECEIVERS AND TALLY MESSAGES

Obviously any video receiver can specify whether the source it is currently on program row or preview row. This is communicated up-stream to the sender of that source, which will then indicate its visibility state (see the section on the sender SDK within this document). The sender takes its current tally state and echoes it back to all receivers as a meta-data message of the form:

```
<ndi_tally_echo on_program="true" on_preview="false"/>
```

This message is very useful so that every receiver can 'know' whether the source it is looking at is on program output. To illustrate this, when a sender is named "My Source A" and is sending to two destinations "Switcher" and "Multi-viewer". When "Switcher" places "My Source A" onto program out, a tally message is sent from "Switcher" to "My Source A". Thus the source now knows it is on program out. At this point it will mirror the tally state as a tally echo to "Multi-viewer" (and "Switcher") so that it is aware that "My Source A" is on program out.

This functionality is used in NDI tools Studio monitor to allow it to display a tally indicator that tells you whether the source being monitored is currently has tally state set.



NDI-ROUTING

Using NDI routing, you can create an output on a machine that looks just like it is a real video source to all remote systems. However, instead of it producing actual video frames, it directs sources watching this output to instead receive their video from a different location.

For instance: if you have two NDI video sources, "Video Source 1" and "Video Source 2", you can create an `NDI_router` called "Video Routing 1", and direct it at "Video Source 1". "Video Routing 1" will be visible to any NDI receivers on the network as an available video source. When receivers connect, the data they receive will be from "Video Source 1".

NDI routing does not actually transfer any data through the computer hosting the routing source; it merely instructs receivers to look at another location when they wish to receive data from the router. Thus a computer can act as a router exposing potentially hundreds of routing sources to the network without any bandwidth overhead. This facility can be used for large scale dynamic switching of sources at a network level.

You create a video routing source using:

```
NDIlib_routing_instance_t NDIlib_routing_create(  
    const NDIlib_routing_create_t* p_create_settings);
```

The creation settings allow you to assign a name and group to the source that is created. Once the source is created, you can tell it to route video from another source using :

```
bool NDIlib_routing_change(NDIlib_routing_instance_t p_instance,  
    const NDIlib_source_t* p_source);
```

and :

```
bool NDIlib_routing_clear(NDIlib_routing_instance_t p_instance);
```

Finally, when you are finished, you can dispose of the router using:

```
void NDILib_routing_destroy(NDILib_routing_instance_t p_instance);
```

FRAME TYPES

Sending and receiving use common structures to define video, audio and metadata types. The parameters of these structures are documented below.

VIDEO FRAMES (NDILIB_VIDEO_FRAME_V2_T)

xres, yres (int)

This is the resolution of the frame expressed in pixels. For instance, xres=1920, yres=1080 could specify a 1080i or 1080p video frame. Note that, because data is internally all considered in 4:2:2 formats, image width values should be divisible by two.

FourCC (NDILib_FourCC_type_e)

This is the pixel format for this buffer. There are currently two supported formats, as listed in the table below.

FourCC	Description
NDILib_FourCC_type_UYVY	YUV 4:2:2 (Y sample at every pixel, U and V sampled at every second pixel horizontally on each line). A macro-pixel contains 2 pixels in 1 DWORD. Please see notes below regarding the expected YUV color space for different resolutions. Note that when using UYVY video, the color space is maintained end-to-end through the pipeline, which is consistent with how almost all video is created and displayed.
NDILib_FourCC_type_YV12	Planar 4:2:0 YUV video data in Y, U, V ordering in memory.
NDILib_FourCC_type_I420	Planar 4:2:0 YUV video data in Y, V, U ordering in memory.
NDILib_FourCC_type_NV12	Planar 4:2:0 YUV video in a Y plane and then an interleaved UV plane.
NDILib_FourCC_type_BGRA	BGRA linear video. This data is not pre-multiplied.
NDILib_FourCC_type_BGRX	BGRX linear video. This is identical to BGRA but is provided as a hint that all alpha channel values are 255, meaning that alpha compositing may be avoided. The lack of an alpha channel is used by the SDK to improve performance.
NDILib_FourCC_type_RGBA	RGBA linear video. This data is not pre-multiplied.
NDILib_FourCC_type_RGBX	RGBX linear video. This is identical to RGBX but is provided as a hint that all alpha channel values are 255, meaning that alpha compositing may be avoided. The lack of an alpha channel is used by the SDK to improve performance.

When running in a YUV color space, the following standards are applied:

Resolution	Standard
SD resolutions	BT.601
HD resolutions	Rec.709

UHD resolutions	Rec.2020
Alpha channel	Full 0-255 range when running 8 bit.

For the sake of compatibility with standard system components, Windows APIs expose 8 bit UYVY and RGBA video (common FourCCs used in all media applications).

frame_rate_N, frame_rate_D (int)

This is the framerate of the current frame. The framerate is specified as a numerator and denominator, such that the following is valid:

```
frame-rate = frame_rate_n/frame_rate_d
```

Some examples of common framerates are presented in the table below.

Standard	Frame-rate ratio	Frame-rate
NTSC 1080i59.94	30000 / 1001	29.97Hz
NTSC 720p59.94	60000 / 1001	59.94Hz
PAL 1080i50	30000 / 1200	25Hz
PAL 720p50	60000 / 1200	50Hz
NTSC 24fps	24000 / 1001	23.98Hz

picture_aspect_ratio (float)

The SDK defines picture aspect ratio (as opposed to pixel aspect ratios). Some common aspect ratios are presented in the table below. When the aspect ratio is 0.0 then it is interpreted as xres/yres, or that the pixels are square; for most modern video types this is a default that can be used.

Aspect Ratio	Calculated as	image_aspect_ratio
4:3	4.0/3.0	1.333...
16:9	16.0/9.0	1.667...
16:10	16.0/10.0	1.6

is_progressive (NDllib_frame_format_type_e)

This is used to determine the frame type. Possible values are listed in the next table.

NDllib_frame_format_type_progressive	This is a progressive video frame
NDllib_frame_format_type_interleaved	This is a frame of video that is comprised of two fields. The upper of those fields comes first and the lower comes second (see note below)
NDllib_frame_format_type_field_0	This is an individual field 0 from a fielded video frame. This is the first temporal, upper field (see note below).

NDIlib_frame_format_type_field_1	This is an individual field 1 from a fielded video frame. This is the second temporal, lower field (see note below).
----------------------------------	--

To make everything as easy to use as possible, the SDK always assumes that fields are ‘top field first’. This is, in fact, the case for every modern format, but does create a problem for two specific older video formats as discussed below:

NTSC 486 lines

The best way to handle this format is simply to offset the image by one line (`p_uyvy_data + uyvy_stride_in_bytes`) and reduce the vertical resolution to 480 lines. This can all be done without modification of the data being passed in at all: simply change the data and resolution pointers.

DV NTSC

This format is a relatively rare these days, although still used from time to time. There is no entirely trivial way to handle this other than to move the image down one line and add a black line at the bottom.

timecode (int64_t, 64bit signed integer)

This is the timecode of this frame in 100ns intervals. This is generally not used internally by the SDK, but is passed through to applications, which may interpret it as they wish. When sending data, a value of `NDIlib_send_timecode_synthesize` can be specified (and should be the default). The operation of this value is documented in the sending section of this documentation.

p_data (const uint8_t*)

This is the video data itself laid out linearly in memory in the FourCC format defined above. The number of bytes defined between lines is specified in `line_stride_in_bytes`. No specific alignment requirements are needed, although larger data alignments might result in higher performance (and the internal SDK codecs will take advantage of this where needed).

line_stride_in_bytes (int)

This is the inter-line stride of the video data, in bytes.

p_metadata (const char*)

This is a per frame meta-data stream that should be in UTF8 formatted XML and NULL terminated. It is sent and received with the frame.

timestamp(int64_t, 64bit signed integer)

This is a per-frame timestamp filled in by the NDI SDK using a high precision clock. It represents the time (in 100ns intervals measured in UTC time, since the Unix Time Epoch 1/1/1970 00:00) when the frame was submitted to the SDK. On modern sender systems this will have ~1uS accuracy; this can be used to synchronize streams on the same connection, between connections and between machines. For inter-machine synchronization, it is important to use external clock locking capability with high precision (such as NTP).

AUDIO FRAMES (NDILIB_AUDIO_FRAME_V2_T)

NDI Audio is passed to the SDK in floating point and has a dynamic range that is without practical limits without clipping. In order to define how floating point values map into real-world audio levels, a sine-wave that is 2.0 floating point units peak-to-peak (i.e. -1.0 to +1.0) is assumed to represent an audio level of +4dBu corresponding to a nominal level of 1.228V RMS. In order to help explain the relationship between different audio levels, below are two tables that explain the

relationship between NDI audio values for the SMPTE and EBU audio standards. In general we strongly recommend that you use the NDI tools which include “Pattern Generator” and “Studio Monitor” that provide correct audio calibration for different audio standards and you verify that your implementation matches these.

SMPTE AUDIO LEVELS					Reference Level	
NDI	0.0	0.063	0.1	0.63	1.0	10.0
dBu	-∞	-20dB	-16dB	+0dB	+4dB	+24dB
dBVU	-∞	-24dB	-20dB	-4dB	+0dB	+20dB
SMPTE dBFS	-∞	-44dB	-40dB	-24dB	-20dB	+0dB

If you want a "simple recipe" that matches SDI audio levels based on the SMPTE audio standard, you would want to have 20dB of headroom above the SMPTE reference level at +4dBu, which is at +0dBVU, at a level of 1.0 in NDI floating point audio. Conversion from floating point to integer audio would thus be performed with:

```
int smpte_sample_16bpp = max(-32768, min(32767, (int)(3276.8f*smpte_sample_fp)));
```

EBU AUDIO LEVELS					Reference Level	
NDI	0.0	0.063	0.1	0.63	1.0	5.01
dBu	-∞	-20dB	-16dB	+0dB	+4dB	+18dB
dBVU	-∞	-24dB	-20dB	-4dB	+0dB	+14dB
EBU dBFS	-∞	-38dB	-34dB	-18dB	-14dB	+0dB

If you want a "simple recipe" that matches SDI audio levels based on the EBU audio standard, you would want to have 18dB of headroom above the EBU reference level at 0dBu; i.e. 14dB above the SMPTE/NDI reference level. Conversion from floating point to integer audio would thus be performed with:

```
int ebu_sample_16bpp = max(-32768, min(32767, (int)(6540.52f*ebu_sample_fp)));
```

Because many applications provide interleaved 16bpp audio, the NDI library includes utility functions that will convert in and out of floating point formats from PCM 16bpp formats. There is also a utility function for sending signed 16 bit audio using *NDIlib_util_send_send_audio_interleaved_16s*. We would refer you to the example projects, and also the header file *Processing.NDI.utilities.h* which lists the functions available. In general, we recommend the use of floating point audio since clamping is not possible and audio levels are well defined without a need to consider audio headroom.

The audio sample structure is defined as described below.

sample_rate (int)

This is the current audio sample rate. For instance, this might be 44100, 48000 or 96000. It can, however, be any value.

no_channels (int)

This is the number of discrete audio channels. 1 represents MONO audio, 2 represents STEREO, and so on. There is no reasonable limit on the number of allowed audio channels.

no_samples (int)

This is the number of audio samples in this buffer. Any number and will be handled correctly by the NDI SDK. However, when sending audio and video together, please bear in mind that many audio devices work better with audio buffers of the same approximate length as the video framerate. We encourage sending audio buffers that are approximately half the length of the video frames, and that receiving devices support buffer lengths as broadly as they reasonably can.

timecode (int64_t, 64bit signed integer)

This is the timecode of this frame in 100ns intervals. This is generally not used internally by the SDK, but is passed through to applications who may interpret it as they wish. When sending data, a value of `NDILib_send_timecode_synthesize` can be specified (and should be the default), the operation of this value is documented in the sending section of this documentation. `NDILib_send_timecode_synthesize` will yield UTC time in 100ns intervals since the Unix Time Epoch 1/1/1970 00:00. When interpreting this timecode a receiving application may choose to localise the time of day based on time zone offset which can optionally be communicated by the sender in connection metadata. Since timecode is stored in UTC within NDI, communicating timecode time of day for non UTC time zones requires a translation.

`p_data` (const float*)

This is the floating point audio data in planar format with each audio channel stored together with a stride between channels specified by `channel_stride_in_bytes`.

`channel_stride_in_bytes` (int)

This is the number of bytes that are used to step from one audio channel to another.

`p_metadata` (const char*)

This is a per frame meta-data stream that should be in UTF8 formatted XML and NULL terminated. It is sent and received with the frame.

`timestamp`(int64_t, 64bit signed integer)

This is a per-frame timestamp filled in by the NDI SDK using a high precision clock. It represents the time (in 100ns intervals measured in UTC time, since the Unix Time Epoch 1/1/1970 00:00) when the frame was submitted to the SDK. On modern sender systems this will have ~1uS accuracy and can be used to synchronize streams on the same connection, between connections and between machines. For inter-machine synchronization it is important that some external clock locking capability with high precision is used, such as NTP.

METADATA FRAMES (NDILIB_METADATA_FRAME_T)

Meta data is specified as NULL-terminated, UTF8 XML data. The reason for this choice is so the format can naturally be extended by anyone using it to represent data of any type and length. XML is also naturally backwards and forwards compatible, because any implementation would happily ignore tags or parameters that are not understood, which in turn means that devices should naturally work with each other without requiring a rigid set of data parsing and standard complex data structures.

If you wish to put your own vendor specific metadata into fields, please use XML namespaces. The “NDI” XML namespace is reserved.

It is very important that you compose legal XML messages for *sending*. (On *receiving* metadata, it is important that you support badly-formed XML in case a sender did send something incorrect.)

If you want specific meta-data flags to be standardized, please contact us.

`length` (int)

This is the length of the timecode in UTF8 characters. It includes the NULL terminating character. If this is zero then the length will be derived from the string length automatically.

`timecode` (int64_t, 64bit signed integer)

This is the timecode of this frame in 100ns intervals. It is generally not used internally by the SDK, but is passed through to applications who may interpret it as they wish. When sending data, a value of `NDILib_send_timecode_synthesize` can be specified (and should be the default), the operation of this value is documented in the sending section of this documentation.

`p_data (const char*)`

This is the XML message data.

DIRECTSHOW FILTER

The windows version of the NDI SDK includes a DirectShow audio and video filter. This is particularly useful for people wishing to build simple tools, and integrate NDI video into WPF applications.

Both x86 and x64 versions of this filter included in the SDK. If you wish to use them, you must first register those filters using `regsvr32`. The SDK install will register these filters for you. The redistributable NDI installer will also install and register these filters, and can be downloaded by users from <http://new.tk/NDIRedistV3>. You may of course include the filters in your own application installers under the terms of the NDI license agreement.

Once the filter is registered, you can instantiate it by using the GUID:

```
DEFINE_GUID(CLSID_NdiSourceFilter, 0x90f86efc, 0x87cf, 0x4097,  
            0x9f, 0xce, 0xc, 0x11, 0xd5, 0x73, 0xff, 0x8f);
```

The filter name is “NewTek NDI Source”. This filter will present audio and video pins that you may connect to. Audio is supported in floating point and 16bit, video is supported in UYVY and BGRA.

This filter can be added to a graph and will respond to the `IFileSourceFilter` interface. This interface will take “filenames” of the form:

```
ndi://computername/source
```

This will connect to the “source” on a particular “computer name”. For instance, to connect to an NDI source called “MyComputer (Video 1)”, you need to escape the characters and use the following URL:

```
ndi://MyComputer/Video+1
```

Due to the fact that NDI sends frames when needed, not at a constant rate and that there is no way to know if a source will send audio or video in advance, you will need to use two instances to receive both audio and video.

For example, DShow would freeze the graph waiting for audio to match the video frame it received, but the sender may never send audio. Therefore you must treat the audio and video as separate live sources.

With no options added, the filter will default to video. This is equivalent to adding the `video=true` option.

```
NDI://computername/source?video=true
```

To receive the audio stream, use the `audio=true` option on a second instance.

```
NDI://computername/source?audio=true
```

If both audio and video are set to true on the same instance, then `audio=true` will be ignored.

Additional options may be specified in the standard method that they are added to URLs. For example:

```
NDI://computername/source?low_quality=true  
NDI://computername/source?video=true&low_quality=true&force_aspect=1.33333&rgb=true
```

FFMPEG SUPPORT

We have received a lot of communication asking if we can help provide a version of FFMPEG that integrates the NDI support that is not part of the standard FFMPEG builds (when the option is enabled). For this reason we have included the Windows compiled versions of FFMPEG within the SDK as a matter of convenience. The NDI support within FFMPEG was not authored by NewTek and we cannot directly support it although we will always do our best to help.

LICENSING

The FFMPEG DLLs provided are compiled under the GPL license which is available online at <https://www.ffmpeg.org/legal.html>. NewTek strongly support and appreciate the work done by the FFMPEG team and encourage any users to ensure that they fully understand this license and how any FFMPEG related code can be used within their applications. NewTek also sincerely acknowledge and thank Max Verem who implemented the NDI support within FFMPEG entirely and donated it to the community, entirely independently of NewTek. He has consistently supported and worked with the open source community on a number of products of great value to the industry.

FINDING VIDEO SOURCES

The first step if using stand-alone FFMPEG is going to be to locate all network video sources. This can be done with the following command line.

```
ffmpeg -f libndi_newtek -find_sources 1 -i dummy
```

This will provide a list onto standard output of the lists of possible NDI available sources.

RECEIVING NDI SOURCES

With knowledge of the network names, you can now use them as a standard input to any FFMPEG command line. For instance you might be able to encode an NDI input named "SIRIUS (OUT 1)" into an H.264 file with a command line as follows:

```
ffmpeg -f libndi_newtek -i "SIRIUS (OUT 1)" -c:v libx264 -preset fast -c:a copy output.mkv
```

Bear in mind that since FFMPEG can stream video to the internet, act as an RTSP source and much more, you can use it as a tool to build almost any type of video piping using NDI that you wish.

If you wish to encode for the internet and ensure that video is always progressive then you can specify an additional parameter to enable this:

```
ffmpeg -f libndi_newtek -allow_video_fields 0 -i "SIRIUS (OUT 1)" -c:v libx264 -preset fast  
-c:a copy output.mkv
```

SENDING NDI SOURCES

If you have a RTSP camera on your network and you wish to make it an NDI source, it is important to make sure that FFMPEG is converting it into the UYVY color space so that the NDI plugin can process it. It is also generally best to specify an audio format of planar floating point audio. An example that would take a camera in this format and make it visible as an NDI source named "RTSP Camera" would be achieved as follows

```
ffmpeg -i rtsp://10.28.1.168:8557/h264 -pix_fmt uyvy422 -sample_fmt fltp -f libndi_newtek  
-y "RTSP Camera"
```

If you now want to play from a file, it is going to be important that audio and video are clocked to real time, since FFMPEG by default will process as fast as it can. Here you would add parameters that would enable video and audio clocking.

```
ffmpeg -i a_cool_file.mp4 -clock_video 1 -clock_audio 1 -pix_fmt uyvy422 -sample_fmt fltp  
-f libndi_newtek -y "A cool file !"
```

While clocking is almost always needed for file playback, it might at times also be important even when using network sources if the “jitter” on the incoming stream is sufficiently large that the timing is not sufficiently smooth that it would look good on a video output.

VIDEO QUALITY AND PERFORMANCE

On a typical modern i7 computer system, the codec is able to compress 1920x1080 at about 250 frames per second using one CPU core. A benchmark is included in the SDK for you to test your machine.

The PSNR of the codec exceeds 70dB on typical video content. A unique property of this implementation is that – once it enters the compressed video space - any further compression do not incur any further generational loss. For instance, the following image shows the first generation, the second generation of decompression then re-compression, and then the 1000th generation (!) of decompression and re-compression.

Original Image



Generation 1



Generation 1000



Because the compression is 'local', even mixing compressed video frames from different sources does not incur additional 'generation losses'.

The codec is designed to run very fast on modern PCs, and is largely implemented in hand-written assembly. Generally, compressing video frames uses no more CPU resources than a typical capture card in a system might require. In general the 64bit implementation performance is preferred when there is an option.

CONVENTIONS AND NOTES

STRING FORMATS

All strings are defined as being NULL terminated UTF8 strings. It is important that you convert from your local character format type into (and out of) UTF8 where relevant. This format is used since it is the most widely adopted and best practical format for representation of international strings.

TIME-CODE

All messages of all types within NDI have an associated time-code value, defined as a 64 bit integer representing a time in 100ns intervals.

LATENCY

Latency of NDI is better than one field. Because the SDK implementation typically provides frame-at-a-time for reasons of compatibility with existing systems, the minimum latency is likely to be one frame. A hardware implementation can provide full end-to-end latency within 16 scan-lines.

CONTACT DETAILS

We take any bugs seriously and are happy to help support you as best we possibly can. You can email any bug reports or problems to ndi@newtek.com