NDI®

SDK VERSION 4.5.2

4/2/2020

©2010-2019 NewTek Inc.

1 CONTENTS

2 OVERVIEW

The NDI® (Network Device Interface) standard developed makes it easy to prepare products that share video on a local Ethernet network, and includes many additional features and capabilities that have made it by far the world's most prolific broadcast video over IP protocol.

When we first introduced NDI, we stated our conviction that 'the future of the video industry would be one in which video is transferred easily and efficiently in IP space', and that this approach 'would largely supplant current industry-specific connection methods (HDMI, SDI, etc.) in the production pipeline'. By now, the breathtaking transformation we predicted is far advanced, to the extent that hundreds of millions already have NDI-enabled applications at their fingertips.

That a/v signals will be predominantly be carried over IP is no longer in doubt, and vestigial contentions to the contrary have all but sputtered out. All modern video rendering, graphics systems and switchers run on computers; and cameras and most other production devices use computer-based systems internally, too. The vast majority of all such systems are able to communicate via IP – and *NDI is serving this purpose far more often than any other protocol*.

NDI DOESN'T SIMPLY SUBSTITUTE NEWTORK CABLES FOR SDI CABLES – IT CHANGES EVERYTHING!

Handling video over networks opens a world of new creative and pipeline possibilities. 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, it is much more than the sum of its parts. Likewise, introducing video into the IP realm with its endless potential connections has delivered exponential creative possibilities and still expanding 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 very high quality, low latency, frame-accurate video and audio in real time.

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

NDI also includes tools to implement video access rights, grouping, bi-directional metadata, IP commands, routing, discovery servers and more. 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 requiring costly new high-speed network infrastructures. And now, it also revolutionizes ingest and post-production by making fully time-synced capture on a massive scale a reality.

3 CHANGES

Change notes are provided in Section 26 at the end of this manual.

4 LICENSE

You may use the SDK in accordance with the license that is provided with the SDK. 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:

- You may use the NDI library within free or commercial Products (as defined by License) created using this SDK without paying any license fees.
- Your application **must** provide a link to http://ndi.tv/ 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.
- You may not distribute the NDI tools, if you wish to make these accessible to your users you may provide a link to http://ndi.tv/tools/
- NDI is a registered trademark of NewTek and should be used only with the ® as follows: NDI®, along with the statement "NDI® is a registered 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. You are required to use the registered trademark designation only on the first use of the word NDI within a single document.

  Your application's About Box and any other locations where trademark attribution is provided should also specifically indicate that "NDI® is a registered trademark of NewTek, Inc." 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 or 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. Please do not install your NDI dlls into the system path for this reason. If you are distributing the NDI dlls you need to ensure that your application complies with the NDI SDK license, this section and the license terms outlined in "3rd party rights" towards the end of this manual.

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 sdk@ndi.tv.

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.

5 EMBEDDED DEVICE SUPPORT

A number of options are available to provide NDI support to embedded systems or hardware devices. The separate NDI Embedded SDK can be downloaded from the NewTek web site. This SDK includes design details to allow NDI to be compressed on smaller FPGA designs or, in version 4, a method to leverage existing H.264 and AAC encoders

already on a device by simply updating its firmware to support specific requirements (this approach lets you quickly and easily add NDI support to *existing* products).

6 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.

7 HEADER FILES. (NDI_SDK_DIR\INCLUDE*.H)

These files may be distributed with open source projects under the terms of 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.

7.1 BINARY FILES (NDI_SDK_DIR\BIN*.*)

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 section above.

7.2 REDISTRIBUTABLES (NDI_SDK_DIR\REDIST*.EXE)

You are may distribute the NDI redistributables and install them within your own installer. However, you must make all reasonable effort to keep the versions you distribute up to date. You may use the command line with `/verysilent` to install without any user intervention but, if you do, you must ensure that the terms of the NDI license agreement are fully covered elsewhere in your application.

An alternative is to provide a user link to the NewTek provided download of this application at `http://new.tk/NDIRedistV4`. At run-time, the location of the NDI run-time DLLs can be determined from the environment variable `NDI_RUNTIME_DIR_V4`.

7.3 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. Please refer to the guidelines within the "NDI Brand Guidelines" which are included within this folder.

8 LIBRARIES

The NDI SDK includes three individual libraries, as listed below. These share common structures and conventions to facilitate development, and may all be used together.

8.1 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, and any number of receivers can connect to an individual NDI-Send.

8.2 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.

8.3 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 handles all the complexities of reliably receiving high-performance network video.

9 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.

10 COMMAND LINE TOOLS

There are a number of important command line tools within the SDK. There is a discovery server implementation, and a command line application that can be used for recording.

11 CPU REQUIREMENTS

NDI Lib is heavily optimized (much of it is written in assembly). While it detects available architecture and uses the best path it can, the minimum required SIMD level is SSSE3 (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.

Current codecs detect the CPU type at run-time and select the best codec implementation based on the system's capabilities. Current software paths include SSSE3, SSE4, AVX and AVX2 (with and without VEX instructions). Hardware acceleration for certain codecs is now present on Windows and Mac OS, and is supported on Intel, AMD and nVidia based systems.

12 DYNAMIC LOADING OF NDI LIBRARIES

At times you might prefer not to link directly against the NDI libraries, instead loading them dynamically 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 is provided with the SDK.

12.1 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. System dependent #defines are provided to make this a simple process:

- `NDILIB_LIBRARY_NAME` is defined to represent the dynamic library name (as, for example, the dynamic library `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/NDIRedistV4`).

12.2 RECOVERING THE FUNCTION POINTERS

Once you have located the library, you can look for a single exported function `NDIlib_v4_load()`. This function will return a structure of type `NDIlib_v4` that gives you a reference to every NDI function.

12.3 CALLING NDI FUNCTIONS

Once you have a pointer to `NDIlib_v4`, 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( … )`

13 PERFORMANCE AND IMPLEMENTATION

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

13.1 UPGRADING YOUR APPLICATIONS

The libraries (dlls) for NDI v4 should be entirely backwards compatible with NDI v4; you should be able to simply update these in your application to get most of the benefits of the new version without a single code change. The one exception is that with NDI v4.5 you will need to recompile applications for macOS and Linux.

13.2 GENERAL

- Throughout the system, use YCbCr color if possible, as 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.

- 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 continually laying groundwork for these in each new versions so that these will already be in place when the related enhancements 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 more to system memory bandwidth rather than CPU processing performance since the code is designed to keep all execuction pipelines on a CPU busy. NDI takes advantage of multiple CPU cores when decoding and encoding one or more streams and for higher resolutions will use multiple cores to decode a single stream.

- Version 4 of the NDI SDK introduces multi-TCP support which should perform better than other transport mechanisms. It is designed to operate in high performance both in your user mode process (using completion ports on Windows and ePoll on Linux and Mac) and at the kernel level by offloading as much processing to the network card as is possible. We have seen a noticeable CPU performance improvement using this mode as compared to UDP-based modes. (There are some high latency networks in which UDP might perform better.)

- Please email us at sdk@ndi.tv with anything interesting you are doing with the SDK. We are truly interested.

13.3 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 to YCbCr color space for compression. With that said, performance has been significantly improved in version 4 of the NDI SDK.

- Using asynchronous frame submission almost always yields significant performance benefits.

13.4 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.

- 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 resource 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.

13.5 MULTICAST

NDI supports multi-cast based video sources, using multicast UDP with forwards error correction to correct for packet loss. 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; for this reason, multicast sending is disabled by default. Every router that we have tested has treated multicast traffic as if it was broadcast traffic by default. Because most multicast traffic on a network is low bandwidth this is of little consequence, and generally allows a network router to run more efficiently because no packet filtering is required. What this means, though, is that every multicast packet received is sent to *every destination on the network,* regardless of whether it was needed there or not. Because NDI requires high bandwidth multicast, even with a limited number of sources the burden of sending this much data to all network sources on a large network can cripple the entire network's performance.

To avoid this serious problem, it is essential to ensure that *every* router on the network has proper multicast filtering enabled. This option is most commonly referred to as "IGMP snooping". This topic 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 you use multicast NDI with all due caution.

13.5.1 DEBUGGING WITH MULTICAST

Another important cautionary note is that a software application like NDI will subscribe to a multicast group, and will unsubscribe from it when it no longer needs that group.

Unlike most operations in the operating system, the un-subscription step is not automated by the OS; once you are subscribed to a group, your computer will continue to receive data until the router sends an IGMP query to verify whether it is still needed. This happens about every 5 minutes on typical networks.

he result is that if you launch an NDI multicast stream and kill your application *without closing the NDI connection correctly*, your computer will continue to receive the data from the network until this timeout expires.

14 STARTUP AND SHUTDOWN

The commands `NDIlib_initialize()` and `NDIlib_destroy()` can be called to initialize or de-initialize the library. Although never absolutely required, it is recommended that you call these. (Internally all objects are reference-

counted; the libraries are initialized on the first object creation and destroyed on the last, so 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.

15 EXAMPLE CODE

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

UWP Examples	
NDIlib_UWP_GrabStill	This is an example that shows how to use the UWP version of the NDI libraries in order to build an application that could be used on the Universal Windows Platform and would be able to be released on the Microsoft store. There are some important UWP related notes in the Platform considerations section that is next in the manual.
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. Clicking in the window will toggle sending Windows system audio over NDI.
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_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 determiner 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.

16 PLATFORM CONSIDERATIONS

16.1 WINDOWS

The Windows platform is fully supported and provides high performance in all paths of NDI. As with all operating systems, the x64 version provides the best performance. All modern CPU feature sets are supported.

We have found that on some computer systems, when you install "WireShark" to monitor network traffic that it will install a virtual device driver called "NPCap Loopback Driver". This driver will interfere with NDI and potential can cause it to fail to make communications. This is also a potential performance problem for networking since it is designed to intercept network traffic. This driver is not required or used by modern versions of Wireshark. If you find this is installed on your system, then is installed recommend that you go to your network settings and use the context menu on the adapter to disable it.

16.2 WINDOWS UWP

Unfortunately, the Universal Windows Platform has significant restrictions affecting NDI that one needs to be aware of. These are listed below.

- The UWP platform does not allow the receiving of network traffic from Localhost. This means that any sources on your local machine will not be able to be received by a UWP NDI receiver. https://docs.microsoft.com/en-us/windows/iot-core/develop-your-app/loopback
- The current Windows 10 UWP mDNS discovery library has a bug that will not correctly remove an advertisement from the network after the source is no longer available; this source will eventually "time out" on other finders; however this might take a minute or two.
- UWP applications cannot load external DLLs due to sand-boxing, making it unlikely that NDI|HX will work correctly.
- When you create a new UWP project you must ensure you have all of the correct capabilities specified in the manifest for NDI to operate. Specifically at time of writing you need :
    - Internet (Client & Server)
    - Internet (Client)
    - Private Networks (Client & Server)

16.3 MACOS

The Mac platform is fully supported and provides high performance in all paths of NDI. As with all operating systems, the x64 version provides the best performance. Most modern CPU feature sets are supported.

16.4 IOS

iOS supports NDI finding, sending and receiving. Currently the receiving of NDI video streams is not supported, although audio and meta-data works. If you require video decoding then please email us, for some applications we have solutions that might support this.

16.5 LINUX (INTEL)

The Linux version is fully supported and provides high performance in all paths of NDI. The NDI library on Linux depends on two 3rd party libraries:

```
libavahi-common.so.3
libavahi-client.so.3
```

The usage of these libraries depends on the `avahi-daemon` service to be installed and running.

16.6 LINUX (ARM)

The Linux version of NDI, targeted at ARM supports finding, sending and receiving. Currently the receiving of NDI video streams is not supported, although audio and meta-data works. If you require video decoding then please email us, for some applications we have solutions that might support this.

The NDI library on Linux depends on two 3rd party libraries:

```
libavahi-common.so.3
```

```
libavahi-client.so.3
```

The usage of these libraries depends on the `avahi-daemon` service to be installed and running.

17 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 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; the source name is selected by using the current executable name, ensuring that there is a count that ensures sender names are unique (e.g. “My Application”, “My Application 2”, “My Application 3”, etc.)

Supported Parameters	
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. If there group is `NULL` then the system default groups will be used.
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 = nullptr;
create_params_Send.clock_video = true;
create_params_Send.clock_audio = true;

NDIlib_send_instance_t pSend = NDIlib_send_create_v4(&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 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 factor to understand is that video frames are "buffered" on an input; if you provide a video frame to the SDK when there are no current connections to it, the last video frame will automatically be sent when a new incoming connection is received. 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.frame_format_type = 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_v3_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 = nullptr; // No meta-data on this example!

// Submit the buffer
NDIlib_send_send_audio_v4(pSend, &audio_frame);

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

Because many applications like providing interleaved 16bpp audio, the NDI library includes utility functions to convert PCM 16bpp formats to and from floating point formats.

Alternatively, there is a utility function (`NDIlib_util_send_send_audio_interleaved_16s`) for sending signed 16 bit audio. 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 using 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 a metadata message is 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_recv_free_metadata(pSend, &meta_data);
}
```

Connection meta-data, as specified in the NDI-Recv section of this documentation is an important category of meta-data that you will receive automatically as new connections to you are established. This allows an NDI receiver to provide up-stream details to a sender; this could include hints as to what capabilities that the receiver might have. Examples include the resolution and frame-rate preferred by the receiver, its product name, 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 metadata information is handled. 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; it will automatically be sent each time a new connection with the sender is established. The sender internally maintains a copy of any connection metadata messages and sends them automatically.

This is useful to allow a sender to provide downstream information whenever any device 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.
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=\"ABCDEFG\" "
     "             session_name=\"My Midday Show\"/>";

NDIlib_send_add_connection_metadata(pNDI_send, &NDI_connection_type);
```

Because NDI assumes that all senders must have a unique name, and also applies certain filtering to NDI names to make sure that they are network name-space compliant, at times the name of a source you created may be modified slightly. To assist you in getting the exact name of any sender (to ensure you use the same one) there is a function to receive this name.

```
const NDIlib_source_t* NDIlib_send_get_source_name(NDIlib_send_instance_t p_instance);
```

The life-time of the returned value is until the sender instance is destroyed.

17.1 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, nullptr)` 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, nullptr); // Sync here
```

Please note that the most common SDK 'bug report' relates to user error involving asynchronous sending. It is very important to understand that a call to `NDIlib_send_send_video_v2_async` will start processing and then sending the video frame asynchronously with the calling application. If you call this function and then free the pointer, your application will most likely crash in an NDI thread – because the SDK would still be using the video frame that was passed to the call.

If you re-use the buffer immediately after calling this function, your video stream will very likely exhibit tearing or other glitches. This is because you are writing to the buffer while the SDK is still compressing the data it held previously. One possible solution is to "ping pong" between two buffers on alternating calls to `NDIlib_send_send_video_v2_async`, and then call that same function with a `null` frame pointer before releasing these buffers at the end of your application. When working in this way you would generally render, compress and send to the network, with each process asynchronous being to the others.

17.2 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`) to cause the SDK to generate timecode for you. When you specify this, the timecode is synthesized 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, the current system clock time is used as the starting timecode (translated to UTC since the Unix Epoch), and synthetic values are generated, thus 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 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 generated will continue this sequence. This maintains the correct relationship between the

streams and samples generated, avoiding deviations from the timecode that you specified over time in any meaningful way.

If you specify timecodes on one stream (e.g. video) and ask for the other stream (audio) to be synthesized, the timecodes generated for the other stream exactly match the correct sample positions; they are not quantized inter-stream. 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; if there is no sample that looks close, a timecode is synthesized from the last ones known and the time that has 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 have the same timecode, even though the video frame may have taken a few milliseconds longer 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.

17.3 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 nullptr, the failsafe source will be cleared.

17.4 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, we recommend releasing an NDI sender instance within the app's `applicationDidEnterBackground` method, then recreating the instance in the `applicationDidBecomeActive` method.

18 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 `nullptr` creation parameter in which case default parameters are used. If you wish to specify the parameters manually, then the member values are as follows:

Supported Values	
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 nullptr 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 = nullptr;

// 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 you 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`.

19 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 nullptr and default settings are used. This takes parameters defined by `NDIlib_recv_create_v3_t`, as follows below:

Supported Parameters	
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 nullptr 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 nullptr 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 table lists the optional values that can be used to specify the color format to be returned.

Optional color_format values	Frames without alpha	Frames with alpha
NDIlib_recv_color_format_BGRX_BGRA	BGRX	BGRA
NDIlib_recv_color_format_UYVY_BGRA	UYVY	BGRA
NDIlib_recv_color_format_RGBX_RGBA	RGBX	RGBA
NDIlib_recv_color_format_UYVY_RGBA	UYVY	RGBA
NDIlib_recv_color_format_fastest	Normally UYVY. See notes below.	Normally UYVA. See notes below.
NDIlib_recv_color_format_best	Varies. See notes below.	Varies. See notes below.

color_format notes:

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. On most platforms this will return an 8bit UYVY video buffer when there is no alpha channel, and an 8bit UYVY+A buffer when there is. These formats are described in the description of the video layout.

If you specify the color option `NDIlib_recv_color_format_best`, the SDK will provide you buffers in the format closest to the native precision of the video codec being used. In many cases this is both high-performance and high-quality and results in the best quality. Like the `NDIlib_recv_color_format_fastest`, this format will always deliver individual fields, implicitly assuming the `allow_video_fields` option as true.

On most platforms when there is no alpha channel, this will return either a 16bpp Y+Cb,Cr (P216 FourCC) buffer when the underlying codec is native NDI and a 8bpp UYVY buffer when the native codec is an 8bit codec like H.264. When there is alpha channel, this will normally return a 16bpp Y+Cb,Cr+A (PA16 FourCC) buffer.

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.

Supported Parameters (Continued)	
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 nullptr-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 nullptr 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 nullptr 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_v3_t`), video (`NDIlib_video_frame_v2_t`) and metadata (`NDIlib_metadata_frame_t`), any of which can be nullptr. It 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 `NDIlib_recv_capture` function takes a timeout value specified in milliseconds. If you call `NDIlib_recv_capture` and a frame is available, 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_v3_t audio_frame;
NDIlib_metadata_frame_t metadata_frame;
```

```
switch(NDIlib_recv_capture_v4(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_v4(pRecv, &video_frame);
           break;

    // We received audio.
    case NDIlib_frame_type_audio:
           // Process audio here
           // Free the audio.
           NDIlib_recv_free_audio_v4(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 most convenient for 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.

Either `NDIlib_recv_capture_v4` or `NDIlib_recv_capture` may return the value `NDIlib_frame_type_status_change`, to indicate that the device's properties 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 this does become known, the value `NDIlib_frame_type_status_change` is returned to indicate that you should recheck device properties. This value 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 supply 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 functions provided by the receive SDK allow 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=\"ABCDEFG\"/>";

NDIlib_recv_add_connection_metadata(pNDI_recv, &NDI_connection_type);
```

19.1 RECEIVER USER INTERFACES

A sender might provide an interface that allows configuration. For instance a NDI converter device might offer an interface that allows its settings to be changed, 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 example, 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_recv_get_web_control(NDIlib_recv_instance_t p_instance);
```

This will return a string representing the URL, or nullptr 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, note that `NDIlib_recv_capture_v4` and `NDIlib_recv_capture` both return a value of `NDIlib_frame_type_status_change` when this setting is known (or when it has changed).

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

```
const char* p_url = NDIlib_recv_get_web_control(p_NDIRecv);
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_recv_free_string(p_NDIRecv, 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 opens the web page specified by the sender.

19.2 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 on-screen PTZ controls when the current source is reported to be a camera that supports control.

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 connections are not instantaneous, so you might need to wait a few seconds after connection in order for the source to indicate that it supports PTZ control. To avoid the need to poll this setting, note that `NDIlib_recv_capture_v4` and `NDIlib_recv_capture` both return a value of `NDIlib_frame_type_status_change` when this setting is known (or when it has changed).

19.2.1 PTZ CONTROL

There are standard API functions to execute the standard set of PTZ commands. This list is not designed to be exhaustive and may 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.

19.2.2 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.

19.2.3 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.

19.2.4 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,

19.2.5 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 NDIlib_recv_ptz_auto_focus(NDIlib_recv_instance_t p_instance);
bool NDIlib_recv_ptz_focus(NDIlib_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 NDIlib_recv_ptz_focus_speed(NDIlib_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.

19.2.6 WHITE BALANCE

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

```
bool NDIlib_recv_ptz_white_balance_auto(NDIlib_recv_instance_t p_instance);
```

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

```
bool NDIlib_recv_ptz_white_balance_indoor(NDIlib_recv_instance_t p_instance);
```

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

```
bool NDIlib_recv_ptz_white_balance_outdoor(NDIlib_recv_instance_t p_instance);
```

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

```
bool NDIlib_recv_ptz_white_balance_manual(NDIlib_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 NDIlib_recv_ptz_white_balance_oneshot(NDIlib_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.

19.2.7 EXPOSURE CONTROL

Exposure can either be automatic or manual.

```
bool NDIlib_recv_ptz_exposure_auto(NDIlib_recv_instance_t p_instance);
```

This will place the camera in auto exposure mode.

```
bool NDIlib_recv_ptz_exposure_manual(NDIlib_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].

19.3 RECEIVERS AND TALLY MESSAGES

Any video receiver can specify whether the source is currently on a video switcher's program row or preview row. This is communicated up-stream to the source's sender, which then indicates 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, allowing every receiver to 'know' whether its source is on program output. To illustrate this, consider a sender named "My Source A" that 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 itself now 'knows' it is visible on program output. At this point, it will echo its tally state to "Multi-viewer" (and "Switcher"), so that the receiver is aware that "My Source A" is on program out.

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

19.4 FRAME SYNCRONIZATION

When using video, it is important to realize that often you are using different clocks for different parts of the signal chain.

Within NDI, the sender can send at the clock rate it wants, and the receiver will receive it at that rate. In many cases, however, the sender and receiver are extremely unlikely to share the *exact same* clock rate. Bear in mind that computer clocks rely on crystals which – while notionally rated for the same frequency – are seldom truly identical.

For example, your sending computer might have an audio clock it rated to operate at 48000Hz. It might well actually run at 48001Hz, or perhaps 47998Hz, however. And similar variances affect receivers. While the differences appear miniscule, they accumulate to cause audio sync to drift over time. A receiver may receive more samples than it plays back; or audible glitches can occur because too few audio samples are sent in a given timespan. Naturally, the same problem affects video sources.

It is very common to address these timing discrepancies by having a "frame buffer", and displaying the most recently received video frame. Unfortunately, the deviations in clock-timing prevent this from being a perfect solution. Frequently, for example, video will appear to 'jitter' when the sending and receiving clocks are *almost* aligned (which is actually the most common case).

A "time base corrector" (TBC) or frame-synchronizer for the video clock provides another mechanism to handle these issues. This approach uses hysteresis to determine the best time to either drop or insert a video frame to achieve smooth video playback (audio should be dynamically sampled with a high order resampling filter to adaptively track clocking differences). It's quite difficult to develop something that is correct for all scenarios, so the NDI SDK provides an implementation to help you develop real time audio/video applications without assuming responsibility for the significant complexity involved.

Another way to view what this component of the SDK does is to think of it as transforming 'push' sources (i.e. NDI sources in which the data is pushed from the sender to the receiver) into 'pull' sources, wherein the host application pulls the data down-stream. The frame-sync automatically tracks all clocks to achieve the best video and audio performance while doing so.

In addition to time-base correction operations, the frame sync will also automatically detect and correct for timing jitter that might occur. This internally handles timing anomalies such as those caused by network, sender or receiver side timing errors related to CPU limitations, network bandwidth fluctuations, etc.

A very common application of the frame-synchronizer is to display video on screen timed to the GPU v-sync, in which case you should convert the incoming time-base to the time-base of the GPU. The following table lists some are common scenarios in which you might want to use frame-synchronization:

Scenario	Recommendation
Video playback on screen or a multiviewer	Yes – you want the clock to be synced with vertical refresh. On a multi-viewer you would have a frame-sync for every video source, then call all of them on each v-sync and redraw all sources at that time.
Audio playback through sound card	Yes – the clock should be synced with your sound card clock.
Video mixing of sources	Yes – all video input clocks need to be synced to your output video clock. You can take each of the video inputs and frame-synchronize them together.
Audio mixing	Yes – you want all input audio clocks to be brought into sync with your output audio clock. You would create a frame-synchronizer for each audio source and – when driving the output – call each one, asking for the correct number of samples and sample-rate for your output.
Recording a single channel	No – you should record the signal in the raw form without any re-clocking.
Recording multiple channels	Maybe – If you want to sync some input channels to match a master clock so that they can be ISO-edited, you might want a frame-sync for all sources *except one* (allowing them all to be synchronized with a single channel).

To create a frame synchronizer object, you will call the function below (that is based an already instantiated NDI receiver from which it will get frames). Once this receiver has been bound to a frame-sync, you should use it in order to recover video frames. You can continue to use the underlying receiver for other operations, such as tally, PTZ, meta-data, etc. Remember, iIt remains your responsibility to destroy the receiver – even when a frame-sync is using it (you should always destroy the receiver *after* the framesync has been destroyed).

```
NDIlib_framesync_instance_t NDIlib_framesync_create(NDIlib_recv_instance_t p_receiver);
```

The frame-sync is destroyed with the corresponding call:

```
void NDIlib_framesync_destroy(NDIlib_framesync_instance_t p_instance);
```

In order to recover audio, the following function will pull audio samples from the frame-sync queue. This function will always return data immediately, inserting silence if no current audio data is present. You should call this at the rate that you want audio, and it will automatically using dynamic audio sampling to conform the incoming audio signal to the rate at which you are calling.

Note that you have no obligation to ensure that your requested sample rate, channel count and number of samples match the incoming signal, and all combinations of conversions are supported.

Audio resampling is done with high order audio filters. Timecode and per frame meta-data are inserted into the best possible audio samples. Also, if you specify the desired sample-rate as zero tit will fill in the buffer (and audio data descriptor) with the original audio sample rate. And if you specify the channel count as zero, it will fill in the buffer (and audio data descriptor) with the original audio channel count.

```
void NDIlib_framesync_capture_audio(
NDIlib_framesync_instance_t p_instance, // The frame sync instance
NDIlib_audio_frame_v2_t* p_audio_data,  // The destination audio buffer
const int sample_rate, // Your desired sample rate. 0 for "use source".
const int no_channels, // Your desired channel count. 0 for "use source".
const int no_samples); // The number of audio samples that you wish to get.
```

The buffer returned is freed using the corresponding function:

```
void NDIlib_framesync_free_audio(
         NDIlib_framesync_instance_t p_instance,
         NDIlib_audio_frame_v2_t* p_audio_data);
```

This function will pull video samples from the frame-sync queue. It will always immediately return a video sample by using time-base correction. You can specify the desired field type, which is then used to return the best possible frame.

Note that:

- Field based frame-sync means that the frame-synchronizer attempts to match the fielded input phase with the frame requests so that you have the most correct possible field ordering on output.
- The same frame can be returned multiple times if duplication is needed to match the timing criteria.

It is assumed that progressive video sources can i) correctly display either a field 0 or field 1, ii) that fielded sources can correctly display progressive sources, and iii) that the display of field 1 on a field 0 (or vice versa) should be avoided at all costs.

If no video frame has ever been received, this will return `NDIlib_video_frame_v2_t` as an empty (all zero) structure. This allows you to determine that there has not yet been any video, and act accordingly (for instance you might want to display a constant frame output at a particular video format, or black).

```
void NDIlib_framesync_capture_video(
         NDIlib_framesync_instance_t p_instance,       // The frame-sync instance
         NDIlib_video_frame_v2_t* p_video_data,        // The destination video frame
         const NDIlib_frame_format_type_e field_type); // The frame type that you prefer
```

The buffer returned is freed using the corresponding function:

```
void NDIlib_framesync_free_video(
         NDIlib_framesync_instance_t p_instance,
         NDIlib_video_frame_v2_t* p_video_data);
```

20 NDI-ROUTING

Using NDI routing, you can create an output on a machine that looks just like a 'real' video source to all remote systems. However, rather than producing actual video frames, it directs sources watching this output to receive 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_tp_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);
```

21 COMMAND LINE TOOLS

21.1 RECORDING

In NDI version 4, full, cross platform, native NDI recording is provided as part of the SDK. In order to allow this to be integrated both into end-user applications but scripted environments, this is provided as a command line application. All input and output from this application is provided over `STDIN` and `STDOUT`, allowing you to read and/or write to these in order to control the recorder.

The NDI recording application implements most of the complex components of file recording and may be included in your applications under the NDI SDK license. The functionality provided by the NDI recorder is as follows.

- **Record any NDI source**. For full bandwidth NDI sources no video recompression is performed; the stream is taken from the network and simply stored on disk meaning that a single machine will take almost no CPU usage in order to record streams. File writing uses asynchronous, block file writing which should mean that the only limitation on the number of recorded channels is the bandwidth of your disk sub-system and the efficiency of the system network and disk device drivers.

- **All sources are synchronized**. The recorder will time-base correct all recordings to be locked to the current system clock. This is designed so that if you are recording a large number of NDI sources that the resulting files are entirely synchronized with each-other. Because the files are written with time-code, they may then be used in a nonlinear editor without any additional work required for multi-angle or

multi-source synchronization. If you lock the clock between multiple computers systems using NTP then recordings done independently on all computer systems will automatically always be synchronized.

- **The complexities of discontinuous and unlocked sources are handled correctly**. The recorder will handle cases in which audio and/or video are discontinuous or not on the same clock. It should correctly provide audio and video synchronization in these cases and adapt correctly even when poor input signals are used.

- **High Performance.** By using asynchronous block based disk writing, without any video compression in most cases it means that the number of streams that may be written to disk is largely limited only by your available network bandwidth and the speed of your drives[1]. On a fast system, even a large number of 4K streams may be recorded to disk!

- **Much more …** Having worked with a large number of companies who wish to have recording capabilities we have realized that helping provide a reference implementation that fills in a lot of the edge-cases and problems of recording would be hugely beneficial and by allowing all sources to be synchronized it makes NDI a fundamentally more powerful and useful tool for video in all cases. This is provided cross-platform and may be used under the NDI SDK license in commercial and free applications. Audio is recorded in floating point and so is never subject to audio clipping at record time.

Recording is implemented as a stand-alone executable which allows it either to be used in your own scripting environments (both locally and remotely), but also allows it to be called from an application. The application is designed to take commands in a structured form from `stdin` and put feedback out onto `stdout`.

21.1.1 COMMAND LINE ARGUMENTS

The primary use of the application would be to run it and specify the NDI source name and the destination file-name. For instance, if you wished to record a source called `My Machine (Source 1)` into a file `c:\Temp\A.mov`. The command line to record this would be:

```
"NDI Record.exe" –I "My Machine (Source 1)" –o "c:\Temp\A.mov"
```

This would then start recording when this source has first provided audio and video (both are required so that it knows the format that is needed in the file). Additional command line options are listed below:

Command Line Option	Description
-I "source-name"	**Required option.** The NDI source name to record.
-o "file-name"	**Required option.** The filename you wish to record into. Please note that if the filename exists that it will be appended with a number to ensure that it is unique.
-u "url"	**Optional.** This is the URL of the NDI source if you wish to have recording start slightly quicker, or if the source is not currently visible in the current group or network.

[1] Note that in practice the performance of the device drivers for the disk and network sub-systems quickly become an issue as well. Ensure that you are using well designed machines if you wish to work with large channel counts.

-nothumbnail	**Optional.** Specify whether a proxy file should be written. By default this option is enabled.
-noautochop	**Optional.** When specified this specifies that if the video properties change (resolution, frame-rate, aspect ratio) then it will chop the existing file and start a new one with a number appended. When false it will simply exit when the video properties change, allowing you to start it again with a new file-name should you want. By default if the video format changes then it will simply open a new file in that format without dropping any frames.
-noautostart	**Optional.** This command may be used to achieve frame-accurate recording as needed. When specified, the record application will run and connect to the remote source however it will not immediately start recording. It will them start immediately when you send a `<start/>` message to `stdin`.

Once running, the application can be interacted with by taking input on `stdin`, and will provide response onto `stdout`. These are outlined below.

If you wish to quit the application, the preferred mechanism is described in the input settings section, however one may also press `ctrl+c` to signal an exit and the file will be correctly closed. If you kill the recorder process while it is running the resulting file will be invalid since QuickTime files require an index at the end of the file. The Windows version of the application will also monitor it's launching parent process, and if that should exit then it will correctly close the file and exit.

21.1.2 INPUT SETTINGS

While this application is running, a number of commands can be sent to `stdin`. These are all in XML format and can control the current recording settings. These are outlined as follows.

Command Line Option	Description
<start/>	Start recording at this moment, this is used in conjuction with the "-noautostart" command line.
<exit/> or **<quit/>**	This will cancel recording an exit the moment that the file is completely on disk.
<record_level gain="1.2"/>	This allows you to control the current recorded audio levels in decibels. 1.2 would apply 1.2dB of gain to the audio signal while recording to disk.
<record_agc enabled="true"/>	Enable (or disable) automatic gain control for audio which will use an expander/compressor to normalize the audio while it is being recorded.
<record_chop/>	Immediately stop recording then restart another file without dropping frames.
<record_chop filename="another.mov"/>	Immediately stop recording and start recording another file in potentially a different location without dropping frames. This allows a recording location to be changed on the fly, allowing you to span recordings across multiple drives or locations.

21.1.3 OUTPUT SETTINGS

Output from NDI recording is provided onto `stdout`. The application will place all non-output settings onto `stderr` allowing a listening application to distinguish between feedback and notification messages. For example, in the run log below different colors are used to highlight what is placed on `stderr` (blue) and `stdout` (green).

```
NDI Stream Record v1.00
(c)2019 NewTek, inc.
```

```
[14:20:24.138]: <record_started filename="e:\Temp 2.mov" filename_pvw="e:\Temp
2.mov.preview" frame_rate_n="60000" frame_rate_d="1001"/>
[14:20:24.178]: <recording no_frames="0" timecode="732241356791" vu_dB="-23.999269"
start_timecode="732241356791"/>
[14:20:24.209]: <recording no_frames="0" timecode="732241690457" vu_dB="-26.976938"/>
[14:20:24.244]: <recording no_frames="2" timecode="732242024123" vu_dB="-20.638922"/>
[14:20:24.277]: <recording no_frames="4" timecode="732242357789" vu_dB="-20.638922"/>
[14:20:24.309]: <recording no_frames="7" timecode="732242691455" vu_dB="-17.237122"/>
[14:20:24.344]: <recording no_frames="9" timecode="732243025121" vu_dB="-19.268487"/>
                                                ...
[14:20:27.696]: <record_stopped no_frames="229" last_timecode="732273722393"/>
```

Once recording starts it will put out an XML message that specifies the filename of the recorded file and provide you with the frame-rate. Once recording starts it will then give you the time-code for each recorded frame and the current audio level in decibels. If the audio is silent then the dB level will be `-inf`. If a recording stops it will give you the final timecode written into the file. The time-codes are specified as UTC time since the Unix Epoch (1/1/1970 00:00) with 100nS precision.

21.1.4 ERROR HANDLING

There are a number of different events that can occur which might cause recording errors. The most common of these is that the drive system that you are recording too is not sufficiently fast to record the video data being stored on it or the seek times to write multiple streams end up dominating the performance (note that we do use block writers to avoid this as best is possible). The recorder is designed to never drop frames in a file, however when it cannot write to disk sufficiently fast it will internally "buffer" the *compressed* video until it has fallen about two seconds behind what can be written to disk; meaning that temporary disk or connection performance issues do not damage the recording. Once a true error is detected it will issue a record-error command as follows:

```
[14:20:24.344]: <record_error error="The error message goes here."/>
```

If the option for autochop is enabled, then the recorder will start attempting to write a new file. This process ensures that each file always has all frames without drops, but if there if data needed to be dropped because of insufficient disk performance then the data is missing between files.

21.2 NDI DISCOVERY SERVICE

The NDI discovery service is designed to allow you to replace the automatic discovery that NDI uses with a server that operates as a centralized registry of NDI sources. For installations in which you do not wish to have significant mDNS traffic when you have a large number of sources or in installations in which multicast is not possible[2] or desirable. When using the discovery server then NDI is able to operate entirely in unicast mode and so operate in almost any installation.

The discovery server supports all NDI functionality including NDI groups.

[2] It is very common that cloud computing services do not allow multicast traffic.

21.2.1 SERVER

In order to using a discovery server, it is as simple as running the application in `Bin\Utilities\x64\NDI Discovery Service.exe`. This application will then run a server on your local machine that will accept incoming connections with senders, finders and receivers and coordinate amongst them all to ensure that they are all visible to each other. If you are installing this on a separate machine from the SDK you should ensure that the Visual Studio 2017 C run-time is installed on that machine and that the NDI licensing requirements are met.

A 32bit and 64bit version of the discovery service are available although the 64bit version is recommended. The server will user very little CPU usage although when there are a very large number of source and connections it might use RAM and network traffic between all sources to coordinate source lists. It is of course recommended that you have a static IP address so that any clients that are configured to access it will not lose connections if the IP is dynamically re-assigned.

21.2.2 CLIENTS

Clients should be configured to connect with the discovery server instead of using mDNS to locate sources. When there is a discovery server the SDK will use both mDNS and the discovery server for *finding and receiving* and so will be able to locate sources on the local network that are not on machines configured to use discovery. For *senders*, if a discovery service is specified that mDNS will not be used and so these sources will only be visible to other finders and receivers that are configured to use the discovery server.

21.2.3 CONFIGURATION

In order to configure the discovery server for NDI clients, you may use Access Manager to enter the IP address of the discovery server machine.

22 FRAME TYPES

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

22.1 VIDEO FRAMES (NDILIB_VIDEO_FRAME_V2_T)

Parameter	Description
xres, yres (int)	This is the resolution of the frame expressed in pixels. 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	This is a buffer in the "UYVY" FourCC and represents a 4:2:2 image in YUV color space. There is a Y sample at every pixel, and U and V sampled at every second pixel horizontally on each line. A macro-pixel contains 2 pixels in 1 DWORD. The ordering of these pixels is U0, Y0, V0, Y1. 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_UYVA	This is a buffer that represents a 4:2:2:4 image in YUV color space. There is a Y sample at every pixels with U,V sampled at every second pixel horizontally. There are two planes in memory, the first being the UYVY color plane, and the second the alpha plane that immediately follows the first.
	
	For instance, if you have an image with `p_data` and `stride`, then the planes are located as follows :
	
	`uint8_t *p_uyvy = (uint8_t*)p_data;`
	`uint8_t *p_alpha = p_uyvy + stride*yres;`

NDIlib_FourCC_type_P216	This is a 4:2:2 buffer in semi-planar format with full 16bpp color precision. This is formed from two buffers in memory, the first is a 16bpp luminance buffer and the second is a buffer of U,V pairs in memory. This can be considered as a 16bpp version of NV12.
	
	For instance, if you have an image with `p_data` and `stride`, then the planes are located as follows :
	
	`uint16_t *p_y = (uint16_t*)p_data;`
	`uint16_t *p_uv = (uint16_t*)(p_data + stride*yres);`
	
	As a matter of illustration, a completely packed image would have stride as `xres*sizeof(uint16_t)`.

NDIlib_FourCC_type_PA16	This is a 4:2:2:4 buffer in semi-planar format with full 16bpp color and alpha precision. This is formed from three buffers in memory. The first is a 16bpp luminance buffer, and the second is a buffer of U,V pairs in memory. A single plane alpha channel at 16bpp follows the U,V pairs.
	
	For instance, if you have an image with `p_data` and `stride`, then the planes are located as follows :
	
	`uint16_t *p_y = (uint16_t*)p_data;`
	`uint16_t *p_uv = p_y + stride*yres;`
	`uint16_t *p_alpha = p_uv + stride*yres;`
	
	To illustrate, a completely packed image would have stride as `xres*sizeof(uint16_t)`.

NDIlib_FourCC_type_YV12	This is a planar 4:2:0 in Y, U, V planes in memory.
	
	For instance, if you have an image with `p_data` and `stride`, then the planes are located as follows :
	
	`uint8_t *p_y = (uint8_t*)p_data;`
	`uint8_t *p_u = p_y + stride*yres;`
	`uint8_t *p_v = p_u + (stride/2)*(yres/2);`
	
	As a matter of illustration, a completely packed image would have stride as `xres*sizeof(uint8_t)`.

NDIlib_FourCC_type_I420	This is a planar 4:2:0 in Y, U, V planes in memory with the U, V planes reversed from the YV12 format.
	
	For instance, if you have an image with `p_data` and `stride`, then the planes are located as follows:
	
	`uint8_t *p_y = (uint8_t*)p_data;`
	`uint8_t *p_v = p_y + stride*yres;`
	`uint8_t *p_u = p_v + (stride/2)*(yres/2);`
	
	To illustrate, a completely packed image would have stride as `xres*sizeof(uint8_t)`.

NDIlib_FourCC_type_NV12	This is a semi planar 4:2:0 in Y, UV planes in memory. The luminance plane is at the lowest memory address with the UV pairs immediately following them.
	
	For instance, if you have an image with `p_data` and `stride`, then the planes are located as follows :
	
	`uint8_t *p_y = (uint8_t*)p_data;`
	`uint8_t *p_uv = p_y + stride*yres;`
	
	To illustrate, a completely packed image would have stride as `xres*sizeof(uint8_t)`.

NDIlib_FourCC_type_BGRA	A 4:4:4:4, 8-bit image of red, green, blue and alpha components, in memory order blue, green, red, alpha. This data is not pre-multiplied.
NDIlib_FourCC_type_BGRX	A 4:4:4, 8-bit image of red, green, blue components, in memory order blue, green, red, 255. This data is not pre-multiplied. 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 when possible.
NDIlib_FourCC_type_RGBA	A 4:4:4:4, 8-bit image of red, green, blue and alpha components, in memory order red, green, blue, alpha. This data is not pre-multiplied.
NDIlib_FourCC_type_RGBX	A 4:4:4, 8-bit image of red, green, blue components, in memory order red, green, blue, 255. This data is not pre-multiplied. This is identical to RGBA, 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 possible.

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

Resolution	Standard
SD resolutions	BT.601
HD resolutions `(xres>720 || yres>576)`	Rec.709
UHD resolutions `(xres>1920 || yres>1080)`	Rec.2020
Alpha channel	Full range for data type (0-255 range when running 8bit and 0-65536 range when running 16bit.)

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).

Parameters (Continued)	Description
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

Parameters (Continued)	Description
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

Parameters (Continued)	Description
frame_format_type (NDIlib_frame_format_type_e)	This is used to determine the frame type. Possible values are listed in the next table.

Value	Description
NDIlib_frame_format_type_progressive	This is a progressive video frame
NDIlib_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)
NDIlib_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:

22.1.1 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.

22.1.2 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.

Parameters (Continued)	Description
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 nullptr 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).

22.2 AUDIO FRAMES (NDILIB_AUDIO_FRAME_V3_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.

Two tables are provided below that explain the relationship between NDI audio values for the SMPTE and EBU audio standards. In general we strongly recommend that you take advantage of the NDI tools “Pattern Generator” and “Studio Monitor”, which provide proper audio calibration for different audio standards, to verify that your implementation is correct.

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 20dB of headroom above the SMPTE reference level at +4dBu, which is at +0dBVU, to correspond to 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*. Please refer you to the example projects, and also the header file *Processing.NDI.utilities.h*, which lists the available functions. 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.

Parameter	Description
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 localize 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 nullptr 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.

22.3 METADATA FRAMES (NDILIB_METADATA_FRAME_T)

Meta data is specified as nullptr-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.

Parameter	Description
length (int)	This is the length of the timecode in UTF8 characters. It includes the nullptr terminating character. If this is zero then the length will be derived from the string length automatically.
p_data (const char*)	This is the XML message data.

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.

If you wish to put your own vendor specific metadata into fields, please use XML namespaces. The "NDI" XML name-space is reserved.

Note: 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.

23 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 are 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/NDIRedistV4.

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 "NDI Source". The filter presents audio and video pins you may connect to. Audio is supported in floating point and 16bit, and video is supported in UYVY and BGRA.

The filter can be added to a graph, and will respond to the IFileSourceFilter interface. This interface takes "filenames" in 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
```

To receive just the video stream, use the audio=false option, as follows:

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

Use the video=false option to receive just the audio stream, as in the example below:

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

Additional options may be specified using the standard method to add to URLs, as for example:

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

NDI://computername/source?audio=false&low_quality=true&force_aspect=1.33333&rgb=true

24 3RD PARTY RIGHTS

The NDI libraries make minor use of other third party libraries, for which we are very grateful to the authors. If you are distributing NDI dlls yourself, it is important that your distribution is compliant with the licenses for these third party libraries. For the sake of convenience we have combined these licenses within a single file that you should include, `Processing.NDI.Lib.Licenses.txt` which is included beside the NDI binary files.

25 SUPPORT

Like other areas of the NDI SDK, if you have any problems please email sdk@ndi.tv and we will do our best to support you. Please be aware that our ability to provide performance guidance or debugging on specific machine configurations is going to be limited.

26 CHANGES

26.1 VERSION 4.5.2

- Virtual camera driver allows you to select a forced video mode. This is needed for GotoMeeting that filters cameras based on their resolutions and so might often have not shown our camera simply because of the resolution of the NDI source. By default this is designed to work with GotoMeeting, however the settings are serialized.

26.2 VERSION 4.5.1

- Fix for minor routing visibility issue when using discovery servers and moving connections from private to public groups.

26.3 VERSION 4.5 BETA 2

- Example that shows how to measure the latency of an NDI connection on a single machine. This illustrates the total round-trip time of compression, transmission and decompression.
- Unfortunately, it is not possible to simply replace the “DLL” files on macOS and Linux, you should recompile your application with the latest SDK.
- A note on “npcap loopback adapter” was added under the Windows section of this manual, this driver has been found to interfere with NDI operation and performance.
- Significant improvement to UDP sender that should dramatically improve the performance on many systems and networks. This should reduce the CPU time, and also get far better network efficiency. This will impact unicast and multicast UDP sending with forwards error correction and the local-host special case path.

26.4 VERSION 4.5 BETA 1

- This version completely replaces the multi-TCP implementation. What we have found is that transitions between network infrastructures are very complex to handle well in the real world, for instance having a

10Gbit/s sender and a 1GBit/s when there are many simultaneous connections is a particularly difficult problem because the prioritization of multiple connections is not (in the real world) “fair” which results in some streams getting all of the bandwidth when sharing connection speed at the cost of others. Solving this, while maintaining good performance acceleration (i.e. the NIC doing all the heavy lifting on sending frames and minimizing kernel transitions) is actually a very complex problem. The current version now implements fully jittered packet based sending which chunks that improve DMA performance. The receiver uses back-pressure by carefully choosing when to read on which sockets allowing the NIC itself to implicitly implement bandwidth sharing across multiple simultaneous connections. This worked out way harder than I would have ever imagined. In a fair test, on the same network setup, the new path achieves at least 15% performance improvement over what a typical optimized TCP sending path would achieve (including previous version of the SDK).

- The multi-TCP implementation now better handles cases in which backwards address resolution from receiver to sender cannot easily be established, probably because of address translations. This change also means that fewer connections are established when trying to negotiate the multi-TCP sockets to use. On typical network configurations this will half the number of transient connections made.

- At 1080p60 the number of channels of decoding possible on Windows should increase when you have a larger number of cores available.

- A fix for a problem with unicast UDP sending in situations in which there are multiple simultaneous connections.

- Experimental new H.264 decoder when using hardware acceleration on Windows that should result in better performance by using fully asynchronous decoding using DXVA2. This also supports multiple simultaneous GPUs and spreading the load between them.

- Work around for a fringe issue in Spark+ in preview mode streams.

- Fix for rare issue on Scan Converter that would cause random crash if it was unable to create the loopback audio capture device.

- Fix for latest version of Adobe After Effects plugin in which they now observe the file reader plugin priorities differently and we need to specify these at a different level to allow our file reader to get priority.

- Lower latency in all forms of HX decoding. This is at least 2 frames better in almost all cases and in many it will be better than that.

- Vastly improved startup discovery performance on Windows.

- Improvement that will make “hw_accel” work much better with HX 2 sources, for instance the iOS camera application.

- Configures network stack in a complex way to achieve better TCP performance by avoiding the standard acknowledgment path.

- Fix for problem that would cause RGBA formats to be transposed into BGRA when sending from an NDI 3.x sender to an NDI 4 receiver. This was a receiver side problem so updating to this version of the SDK will solve all problems.

- The managed WpfSendContainer in NdiLibDotNet2 can now optionally send the output of Windows system audio by setting the bool SendSystemAudio to true. It defaults to false to be compatible with existing applications and can be toggled at runtime.
- Quite a large improvement to codec quality.
- Fix for possible lockup when there are a huge number of consecutive “skip blocks” while compressing.
- Significant improvement to network performance based on heuristics we found on real world networks. We expect this to have a noticeable improvement when you are using many streams on the same machine or on a congested network.
- Scan Converter will better handle the cases in which there might be new connections while there is nothing moving on-screen. This is a slightly hard case to handle well because the windows capture APIs (correctly) tell you nothing is changing on screen and so getting passed a new frame is hard.
- Fix for problem with 10bit packed to 16bit conversions.
- Recording now writes floating point audio into the file instead of 24bit integer.
- New recording options while recording are supported to change the audio gain, continue recording on another drive, quickly exit, etc… These are documented in this manual.
- Minor fixes in codec for older version support.
- Example that includes 16bit video support and shows conversion from “V210” (10bit packed format).
- Complete rewrite of the NDI Discovery service so that it will now use asynchronous sending and scatter gather lists to build long connection strings without any memory operations with much lower resources when handling large connection counts.
- Fix for problem with right clicking when using Scan Converter and KVM.
- NDI now uses predictable port numbers in all cases so that fire-wall rules can be properly built.
    - Each connection uses TCP ports 5960 and up.
    - Each mTCP or UDP receiver use port 6960 and up.
    - Each mTCP or UDP sender use port 7960 and up.

26.5 VERSION 4.1

- Rewrite of virtual camera driver to better support edge cases when people open and close it quickly or from multiple apps.
- NDI libraries now no longer need any visual run-time to be installed, making them smaller and more self-standing. This is something that we have wanted to do for some time but it is more complicated in the build process than it should be … and so more “important” features always came ahead of this. Glad that it is finally done though.

26.6 VERSION 4

- Ships!

26.7 VERSION 4 BETA 7 (RC1)

- Significant improvements with multi-TCP sending, particularly on high jitter network. Additional improvements made to avoid cases of packet loss that might have impacted network performance.
- A discovery problem with mDNS taking longer than expected when starting numerous NDI apps at very close to the same time has been resolved.
- Significant improvements to HX decoding performance (both version 1 and version 2) with hardware acceleration. This allows us to get much closer to peak decoder performance in almost all cases on nVidia, AMD and Intel. This applies on both the Mac and Windows platforms.
- New capability to get the number of connections on a routing server using `NDIlib_routing_get_no_connections`.

26.8 VERSION 4 BETA 6

- When sending using multiple NICs we now apply some heuristics to the selection of allowed device pairs.
    - If there is a wireless and a wired network adapter we will not attempt to share bandwidth across both wired and also wireless.
    - If there is a very high bandwidth network adapter and a very low bandwidth adapter we will not attempt to share bandwidth across them.
    - If there is a highly asymmetric bandwidth network (e.g. very high upload, very low download) adapter then we will not use it in bandwidth sharing.

26.9 VERSION 4 BETA 5

- More robust handling of cases in which discovery server goes unexpectedly off-line and then comes online again. A number of fixes for the discovery server when used within poor network environments, when it is killed, or when connections are unexpectedly terminated.
- Crash or garbage on some images in 16bit mode when video was not a multiple of 8 pixels wide. More accurate RGB color conversions and hopefully faster too.
- mDNS improvements for interoperability between Windows and Mac (and potentially other devices).

26.10 VERSION 4 BETA 4

- *Scan Converter include NDI KVM support directly*. You can now run Scan Converter on one machine and Studio Monitor on a second, the second can be used to control the first. Mouse, keyboard, clipboard and touch commands are all supported.

- *Adobe CC File Reading Plugin Included*. A plugin that allows the native SpeedHQ files created by NDI recording is included. This file reader supports clips with and without alpha, any number of audio channels, growing files, automatic time-stamp synchronization to enable multi-cam editing, and much more.

- *Studio Monitor* supports web based control of all settings. Starting and stopping recording is available by web control as well.

- *Studio Monitor* has an NDI output that is a routed version of what is currently selected for view within the monitor.

- *Studio Monitor* web pages have control of the record button.

- The DirectShow filter has been entirely rewritten and allows a simple filter to provide audio and video, supports full audio-video synchronization with dynamic re-clocking. Video format changes are supported (see MSDN documentation on dynamic format changes if you want to support this, https://docs.microsoft.com/en-us/windows/desktop/directshow/dynamic-format-changes).

- The NDI codec decompression has been significantly improved in image quality at no extra performance overhead. This change will benefit all senders whether they are from version 4 (that internally uses an entirely new version of the codec) or version 3.

- Fix for crash in AfterEffects frame-buffer plugin when displaying a clip without it being part of a composition.

26.11 VERSION 4 BETA 3

- The current benchmarks of NDI compression show now massive improvement. Expanded out to count the number of frames encoded per second across multiple threads on an i7-5930 CPU (i.e. an old CPU). This shows massive improvement beyond what we had even expected.

    - NDI 3.0 : 515fps

    - NDI 3.8 : 695fps

    - **NDI 4.0 : 1535fps**

- Fix for very rare crash when receiving video using multi-TCP on connections that have high latency. This could happen every 8 hours or so and so took a long of testing to diagnose and fix.

- Fix and significant performance improvements when decoding RGBA when using new NDI codec that has skip-block elements.

- Better AVX2 codec paths.

- Fix for slow systems using mTCP that are sending data which might lockup when connections get closed.

- The SpeedHQ codecs that are included in NDI tools now have all the performance benefits of using the latest NDI version.

26.12 VERSION 4 BETA 2

- On Windows, automatic discovery of NDI|HX (v1) cameras no longer shows a placeholder image that says “please install the HX driver). This reduces the library sizes, reduces dependencies and also reduces network traffic since detection of HX sources no longer occurs by default with the NDI libraries on their own. We are very hopeful that most systems will end up moving to NDI|HX (v2) over time which is embedded in the SDK and will in most ways operate as efficiently as high-bandwidth NDI feeds, supports mDNS discovery, all full NDI transfer modes, etc…

- Two new utility functions per request, NDIlib_util_V210_to_P216 and NDIlib_util_P216_to_V210. I’m not going to document these beyond this comment but they are there for anyone who wants them ☺

- NDI Studio Monitor has options to flip video horizontally or vertically. This is useful for when talent want to have a preview monitor, or when using a teleprompter.

- Multi-threaded compression and decompression of NDI sources above HD are now enabled again.

- There is an option to disable “Jitter Correction” in Studio Monitor. When this runs, the lowest possible latency is achieved. When in this mode the application will run with the lowest possible latency, although there likely is still some caused by your GPU (swapping he back buffer) and the monitor. Without network jitter correction it is possible that video will look slightly less smooth.

- NDI recording command line application supports the ability to make a connection and get everything ready for recording, but then start actually writing to the file at a moment specified by the user. This allows for frame accurate starting of recording.

- On Windows, the SDK is no longer dependent on Media Foundation libraries and will work when they are not there. We do depend on hardware accelerated capabilities within Media Foundation when using HX version 2, which might mean that on older systems or Windows 10N systems that HX version 2 might not display video. Please note that if this is a problem that you may install the following windows update. https://support.microsoft.com/en-us/help/3099229

26.13 VERSION 4 BETA 1

26.13.1 TOOLS

- Epic Games Unreal 4 plugin.

- Studio Monitor has integrated recording of all NDI sources with all the benefits of NDI recording. By using multiple monitors you can ISO record as many channels.

- Studio Monitor allows you to select the record path for each monitor.

- Video Monitor on Mac includes support for camera registration, PTZ controls and recording.

26.13.2 VIDEO CODEC

- The fastest version of the codec. NDI version 4 is now about 45% faster than the first version of NDI.

- Separate code paths for all common CPU architectures to get best performance.
- On average, 3dB PSNR quality improvements at the same bit-rate.
- Support for full 16bpp FourCCs is provided. The 16bpp code path is highly optimized and does not degrade performance significantly unless memory bandwidth is the greatest performance limitation on your machine.
- RGBA color format performance is much improved.
- RGBA video quality is much improved since intermediary results preserve all 16bits.
- Integrated support for NDI|HX (this is considered NDI|HX version 2).
- Full SDK support for creating NDI|HX version 2 sources. This is provided in the NDI Embedded SDK that is a separate download.
- Hardware acceleration on Windows and Mac for NDI|HX version 2.
- NDI|HX version 2 does not require any external driver to be provided.

26.13.3 AUDIO CODEC

- Integrated support for NDI|HX audio.

26.13.4 DISCOVERY

- Huge improvements to performance, stability and reduced network traffic when using mDNS auto-discovery.
- Support for a discovery service so that you can have a separate server to coordinate and catalog all NDI sources. When using this, multicast is not required for any connections which is important for data center use.

26.13.5 RECORDING

- Record an unlimited number of NDI video channels with all the complexity of matching audio and video time-bases, frame-rates, sample rates, efficient file writing.
- Record any number of NDI video sources without recompression. This means that it takes no CPU time to record any number of channels so that you can trivially iso-record any number of channels; limited only by your disk and network speeds.
- All recordings are time-stamped and time-based corrected so that you can record any number of channels on any number of machines and the results will all be synchronized together.
- Record supports recording with alpha channel.
- Record supports for any number of audio channels.

- Recording works with 16bpp sources at full precision.
- Quicktime and Windows video codecs are provided so you may use these files as needed.
- Adobe CC plugin available for editing NDI recorded files, including growing ones.

26.13.6 TRANSPORT

- New multi-TCP mode that is designed to use hardware accelerated network adapters with adaptive bandwidth sharing across NICs and all network paths, even ones with unmatched bandwidth or other traffic on them. Low kernel overhead and zero memory copy on many platforms.
- High performance IO completion port (Windows) and ePoll (Linux, Mac, IOS) implementations for best performance across all platforms.
- New NDI|HX version 2 includes error resilience by automatically detecting data drops and requesting I-frame insertions so that video will now show artifacts even on high error rate networks.

26.13.7 OPERATING SYSTEMS

- Significantly improved support on macOS and Linux, including support for all transport protocols.