SDK VERSION 5.5

10/19/2022

Table of Contents

1 Overview 4

1.1 NDI Background 4

2 SDK Changes 4

3 Licensing 5

4 Software Distribution 6

4.1 Header files. (NDI_SDK_DIR\Include*.h) 6

4.2 Binary files (NDI_SDK_DIR\Bin*.*) 6

4.3 Redistributables (NDI_SDK_DIR\Redist*.exe) 6

4.4 Content Files (NDI_SDK_DIR\LOGOS*.*) 6

4.5 Libraries 6

4.5.1 NDI-Send 6

4.5.2 NDI-Find 6

4.5.3 NDI-Receive 7

4.6 Utilities 7

4.7 Command line tools 7

5 CPU Requirements 7

6 Dynamic Loading of NDI® Libraries 7

6.1 Locating the Library 7

6.2 Recovering the Function Pointers 8

6.3 Calling NDI functions 8

7 Performance and Implementation 8

7.1 Upgrading Your Applications 8

7.2 General Recommendations 8

7.3 Sending Video 9
7.4 Receiving VIdeo 9
7.5 Reliable UDP With Multi-Stream Congestion Control 9
7.6 Multicast 10
7.6.1 Debugging with multicast 10
8 Startup and Shutdown 11
9 Example Code 11
10 Port Numbers 11
11 Configuration Files 12
12 Platform Considerations 17
12.1 Windows 17
12.2 Windows UWP 17
12.3 macOS 18
12.4 Android 18
12.5 iOS 18
12.6 Linux 18
13 NDI-Send 19
13.1 Asynchronous Sending 22
13.2 Timecode synthesis 23
13.3 Failsafe 24
13.4 Capabilities 24
13.5 Apple iOS Notes 25
14 NDI-Find 25
15 NDI-Recv 27
15.1 Receiver User interfaces 31
15.2 Receiver PTZ Control 32
15.2.1 PTZ Control 32
15.3 Receivers and Tally Messages 34
15.4 Frame Synchronization 34
16 NDI-Routing 37
17 Command Line Tools 38
17.1 Recording 38

17.1.1 Command Line Arguments 39
17.1.2 Input Settings 40
17.1.3 Output Settings 40
17.1.4 Error Handling 41
17.2 NDI Discovery Service 41
17.2.1 Server 41
17.2.2 Clients 42
17.2.3 Configuration 42
17.2.4 Redundancy and multiple servers 42
17.3 NDI Benchmark 42
18 Frame Types 43
18.1 Video Frames (NDIlib_video_frame_v2_t) 43
18.2 Audio Frames (NDIlib_audio_frame_v3_t) 47
18.3 Metadata Frames (NDIlib_metadata_frame_t) 50
19 Windows DirectShow Filter 50
20 3rd Party Rights 51
21 Support 51

1 OVERVIEW

This SDK makes it easy to prepare products that share video on a local Ethernet network, and the countless features and capabilities that have made NDI® by far the world's most prolific broadcast video over IP protocol.

1.1 NDI BACKGROUND

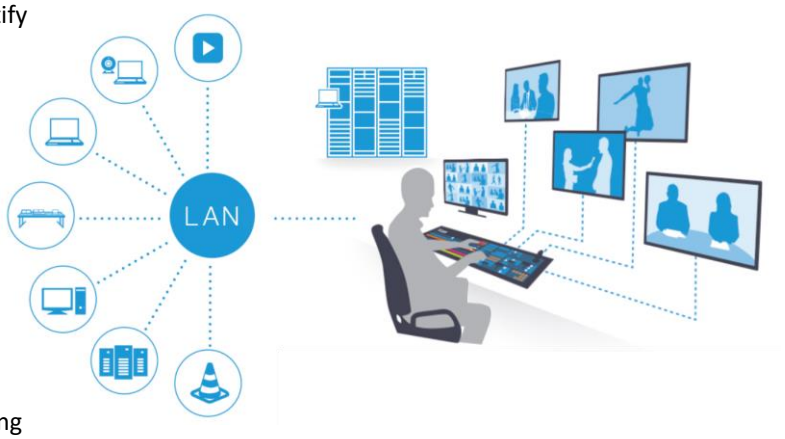
When we first introduced NDI, we made the bold assertion that 'the future of the video industry would be one in which video is transferred easily and efficiently in IP space', and that this approach 'would largely supplant current industry-specific connection methods (HDMI, SDI, etc.) in the production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

That a/v signals will predominantly be carried over IP is no longer in doubt, and vestigial contentions to the contrary are largely being phased out. All modern video rendering, graphics systems and switchers run on computers. Cameras and most other production devices use computer-based systems internally, too. Most of such systems 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 virtually endless potential connections delivers exciting creative new possibilities and ever-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 supports many video streams on a shared local network connection and can operate bi-directionally. Its encoding algorithm is resolution and framerate independent, supports 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 make 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.

Wide distribution within a facility becomes a simple, versatile, and economical reality. And NDI also revolutionizes ingest and post-production by making fully time-synced capture on a massive scale a reality.

2 SDK CHANGES

Change notes are provided in an Appendix at the end of this manual.

3 LICENSING

You may use this SDK in accordance with its License Agreement, which is available for review in the root level of the SDK folder. 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 SDK 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 close to all locations where NDI is used / selected within the product, on your web site, and in its documentation. This will link will point to a landing page that provides information about NDI and access to the tools we provide, along with 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 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 License Agreement, this section and the license terms outlined in "3rd party rights" towards the end of this manual.
- If you are using the FREE NDI SDK on Android then you may build products for mobile devices that may be sold on the online Android stores, for other uses please email sdk@ndi.tv.
- The NDI Advanced SDK is provided to allow anyone to develop products against NDI and to develop without any charge. To use the Advanced SDK in a commercial product or environment you should contact licensing@ndi.tv to receive a vendor ID for your company.

We are very interested in how our technology is being used and would like to maintain a full list of applications using NDI technology. Please let us know about your commercial application (or interesting non-commercial one) using NDI by telling us on our NDI SDK support hub at https://www.ndi.tv/ndiplusdevhub. 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 ensuring that both end-users and developers enjoy a consistent high-quality experience.

NOTE: Because AAC, H.264, and H.265 are formats that potentially are not license free, it is your responsibility to ensure that these are correctly licensed for your product if you are using these with this SDK.

4 SOFTWARE DISTRIBUTION

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 for the SDK.

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.

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

These files may be distributed with open-source projects under the terms of the MIT license and 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.

4.2 BINARY FILES (NDI_SDK_DIR\BIN*.*)

You may distribute these files within your application if your EULA terms cover the specific requirements of the NDI SDK EULA, and your application covers the terms of the License section above.

4.3 REDISTRIBUTABLES (NDI_SDK_DIR\REDIST*.EXE)

You may distribute the NDI redistributable 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 then 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 runtime, the location of the NDI runtime DLLs can be determined from the environment variable `NDI_RUNTIME_DIR_V4`.

4.4 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.

4.5 LIBRARIES

Components included in the SDK provide support for finding (find), receiving (recv), and sending (send). These share common structures and conventions to facilitate development and may all be used together.

4.5.1 NDI-SEND

This is used to send video, audio, and metadata 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 framerate in RGB(+A) and YCbCr color spaces, and any number of receivers can connect to an individual NDI-Send.

4.5.2 NDI-FIND

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

4.5.3 NDI-RECEIVE

This allows you to take sources on the network and receive them. The SDK internally includes all the requisite codecs and handles all the complexities of reliably receiving high-performance network video.

4.6 UTILITIES

To make the library easy to use, the SDK includes several utilities that can be used to convert between common formats. For instance, conversion between different audio formats is provided as a service.

4.7 COMMAND LINE TOOLS

There are also several important command line tools within the SDK, including a discovery server implementation, and a command line application that can be used for recording.

5 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). The NDI library running on ARM platforms requires NEON support. To the degree possible, hardware acceleration of streams uses GPU-based fixed function pipelines for decompression and is supported on Windows, macOS, iOS, tvOS platforms; all GPUs on these platforms are supported with special case optimized support for Intel QuickSync and nVidia. However, this is not required, and we will always fall back to software-based compression and decompression.

6 DYNAMIC LOADING OF NDI® LIBRARIES

At times you might prefer not to link directly against the NDI® libraries, loading them dynamically at run time instead. This can be of value in Open-Source projects.

There is a structure that contains all the NDI entry points for a particular SDK version; calling a single-entry point in the library will recover all these functions. The basic procedure is relatively simple, and an example is provided with the SDK.

6.1 LOCATING THE LIBRARY

You can of course include the NDI runtime within your application folder. Alternatively, on Windows you can install the NDI runtime 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 a C #define to represent the dynamic library name (as, for example, the dynamic library `Processing.NDI.Lib.x64.dll`).
- `NDILIB_REDIST_FOLDER` is a C #define variable that references an environment variable to the installed NDI runtime library (for example, `C:\Program Files\NewTek\NDI Redistributable\)`.
- `NDILIB_REDIST_URL` is a C #define for a URL where the redistributable for your platform may be downloaded (for example, http://new.tk/NDIRedistV5).

On the Mac, it is not possible to specify global environment variables and so there is no standard way for the application to provide to specify a path. For this reason, the redistributable on MacOS is installed within `/usr/local/lib`.

It is our intent to make the process of cross platform loading of the run-times easier for the next version of NDI.

6.2 RECOVERING THE FUNCTION POINTERS

Once you have located the library, you can look for a single exported function `NDIlib_v5_load()`. This function returns a structure of type `NDIlib_v5` that gives you a reference to every NDI function.

6.3 CALLING NDI FUNCTIONS

Once you have a pointer to `NDIlib_v5`, 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(...)`

7 PERFORMANCE AND IMPLEMENTATION

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

7.1 UPGRADING YOUR APPLICATIONS

The libraries (DLLs) for the latest version of NDI should be entirely backwards compatible with NDI v4 and, to the degree possible, even earlier versions. In many cases 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.

Note: There are several exceptions to the statement above, however. If you have software that was built before NDI version 4.5, on macOS and Linux applications for NDI v4.5 you will need to recompile your applications; this change was made because the size of some struct members changed.

7.2 GENERAL RECOMMENDATIONS

- 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 version 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 (256-bit instructions) on Intel platforms and NEON instructions on ARM platforms. Generally, NDI speed limitations relate more to system memory bandwidth than CPU processing performance since the code is designed to keep all execution 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.

7.3 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 yields 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 was significantly improved in version 4 of the NDI SDK.
- Using asynchronous frame submission almost always yields significant performance benefits.

7.4 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_v3` is recommended.

Note that `NDIlib_recv_capture_v3` is multi-thread safe, allowing multiple threads to be waiting for data at once. Using a reasonable timeout on `NDIlib_recv_capture_v3` is better and more efficient than polling it with zero time-outs.

- In the modern versions of NDI there are internal heuristics that attempt to guess whether hardware acceleration would enable better performance. With this said, it is possible to explicitly enable hardware acceleration if you believe that it would be beneficial for your application. This can be enabled by sending an XML metadata message to a receiver as follows:

```
<ndi_video_codec type="hardware"/>
```

- Bear in mind in that decoding resources on some machines are designed for processing a single video stream. In consequence, while hardware assistance might benefit some small number of streams, it may hurt performance as the number of streams increases.
- Modern versions of NDI almost certainly already default to using hardware acceleration in most situations where it would be beneficial and so these settings are not likely to make a significant improvement. In earlier versions concerns around hardware codec driver stability made us less likely to enable these by default, but we believe that this caution is no longer needed.

7.5 RELIABLE UDP WITH MULTI-STREAM CONGESTION CONTROL

In NDI version 5 the default communication mechanism is a reliable UDP protocol that represents the state-of-the-art communication protocol that is implemented by building upon all our experience we have seen in the real world with NDI across a massive variety of different installations. By using UDP it does not rely on any direct round-trip, congestion control or flow control issues that are typically associated with TCP. Most importantly, our observation has been that on real world networks, as you approach the total bandwidth available across many different channels of video that the management of the network of bandwidth becomes the most common problem. Our reliable UDP solves this by moving all streams between sources into a single connection across which the congestion control is applied in aggregate to all streams at once which represents a massive improvement in almost all installations. These streams are all entirely non-blocking with each-other so that even under high packet loss situations that there is no possibility of a particular loss impacting any other portions of the connection.

One of the biggest problems with UDP packet sending is that by needing to send many small packets onto the network that one results in a very large OS kernel overhead. Our implementation works around this by supporting fully asynchronous sending of many packets at once and supporting packet coalescing on the receiving side to allow the kernel and network card driver to offload a huge fraction of the processing.

This protocol also supports high network latencies by having the current best published congestion control systems. This allows many packets to be in flight at a time and very accurately track which packets have been received and adjust the number to the current measured round-trip time.

On Windows where this is supported, receiver side scaling is used to achieve much optimized network receiving that ensures that as the network card interrupts are received that they are always handed off directly to a thread that has an execution context available (which are then directly passed into the NDI processing chain).

For the absolute best performance reliable-UDP supports UDP Segmentation Offload (USO) which allows network interface cards to offload the segmentation of UDP datagrams that are larger than the maximum transmission unit (MTU) of the network medium which can significantly reduce CPU usage.

On Linux, to get the best performance we need to take advantage of Generic Segmentation Offload (GSO) for UDP sending, which might also be referred to as UDP_SEGMENT which was made available in Linux Kernel 4.18. Without this, UDP sending can see significantly increased CPU overhead.

7.6 MULTICAST

NDI supports multicast-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 on a large network, the burden of sending this many data to all network sources 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 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.

7.6.1 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.

The 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.

8 STARTUP AND SHUTDOWN

The functions `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.)

In the latest version of NDI on platforms where the application has permissions it will attempt to configure the firewall settings for the application to allow it to perform video communication over NDI.

The only negative side-effect of this behavior is that more work will be done than is required if you repeatedly create and destroy a single object. 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.

9 EXAMPLE CODE

The NDI® SDK includes many examples to help you get going, including examples that show sending, receiving of data, finding sources, use of 10-bit video, and many more. We recommend that you look closely at these since they provide good, simple illustrations of many possible SDK use cases.

10 PORT NUMBERS

Each NDI connection will require one more port number. Current versions try for these to be in a predictable port range, although if some of this range is taken by other applications it might need to use higher numbers. The following table describes the used port numbers, their types, and their purpose.

It is recommended that you use the connection types that are default for the current version of NDI since these represent the best recommendations that we have tested and observed to yield the best performance in the field. Earlier versions of NDI might use ports in the ephemeral range, although modern versions of NDI no longer use these to ensure that the port numbers are more predictable and easier to configure in

Port number	Type	Use	NDI Version
5353	UDP	This is the standard port used for mDNS communication and is always used for multicast sending of the current sources onto the network.	NDI Version 5

5960 and up	UDP	When using reliable UDP connections it will use a very small number of ports in the range of 5960 for UDP. These port numbers are shared with the TCP connections. Because connection sharing is used in this mode, the number of ports required is very limited and only one port is needed per NDI process running and not one port per NDI connection.	NDI Version 5
5960	TCP	This is an TCP port used for remote sources to query this machine and discover all the sources running on it. This is used for instance when a machine is added by IP address in the access manager so that from an IP address alone all the sources currently running on that machine can be discovered automatically.	NDI Version 5
5961 and up	TCP	These are the base TCP connection used for each NDI stream. For each current connection, at least one port number will be used in this range.	NDI Version 4
6960 and up	TCP/UDP	When using multi-TCP or UDP receiving, at least one port number in this range will be used for each connection.	NDI Version 4
7960 and up	TCP/UDP	When using multi-TCP, unicast UDP, or multicast UDP sending, at least one port number in this range will be used for each connection.	NDI Version 4

11 CONFIGURATION FILES

The NDI® configuration settings are stored in JSON files. The location of these files varies per platform and are described in the next section of the manual.

Please note that when using the Advanced SDK NDI SDK that all settings are per instance and so entirely separate settings may be used for every finder, sender, and receiver; this is incredibly powerful by allowing you to specify per sender or receiver which NICs are used, formats are used, what the machine name is, etc…

Please pay extra attention to the value types, as it is important that these matches what is listed here (e.g., true rather than “true”). Also please note that all these parameters have default values that are the recommended best configuration for most machines, we only suggest you change these values if there is a very specific need for your installation.

```
{
  "ndi": {
    "machinename": "Hello World",
```

This is an option that allows you to change how your machine is identified on the network using NDI, over-riding the local name of the machine. This option should be used with very great care since a clash of machine names on the network is incompatible with mDNS and can cause all other sources to not work correctly. When using this it is essential to ensure that all machine names on the network are unique. We recommend that

avoiding using this parameter when at all possible.

```
"send": {
  "metadata": "<My very cool source/>"
},
```

In version 5 of NDI, it is possible to specify metadata for the current source when creating an NDI sender with the Advanced SDK. When using the NDI finder, it is then possible to receive this metadata in order to receive any number of properties about the source. Please note that you should always make your meta-data in XML format and be careful to correctly escape it when adding it into the JSON configuration file.

```
"networks": {
  "ips": "192.168.86.32,",
  "discovery": "127.0.0.1,127.0.0.1"
},
```

These are extra IP addresses for machines from which you wish to discover NDI sources. Each local machine runs a service on port 5960, which is then connected to by the machine this configuration is run on. This allows sources to be discovered on those IP addresses without needing mDNS to discover it.

Hint: When a Discovery server is used, receivers combine the list of sources found on the discovery server with those discovered via mDNS. Senders however will avoid using mDNS when a discovery server is configured allowing you to run entirely without network multicast if you desire.

Starting in NDI version 5 it is possible to use a comma delimited list of discovery servers for full support for redundancy. For more information, please review the section of the manual regarding the discovery server.

The discovery server list may include port numbers if you do not with the default port of 5959 to be used.

```
"groups": {
  "send": "Public",
  "recv": "Public"
},
```

This is the list of groups that the senders on this system are going to be part of by default. If groups are not specified, senders will be part of the public group by default.

```
" sourcefilter": {
  "regex": "MACHINE .*"
},
```

In NDI version 5, there is the ability to specify a regular expression that will be used to further filter the set of sources that will be visible to NDI finders. This is an advanced option that allows you to specify exactly which sources are going to be visible to the local machine. If your regular expression is not valid, then it will not be applied.

```
"adapters": {
 "allowed": ["10.28.2.10","10.28.2.11"]
},
```

Starting in NDI version 5, this lists all the network

adapters that will be used for network transmission. One or more NICs can be used for transmission and receipt of video and audio data. This capability can be used to ensure that the NDI primary stream data remains on group of network adapters, for instance allowing you to ensure that dedicated audio is on a separate network card from the NDI video.

It is generally preferred that you let NDI select the network adapters automatically which can smartly select which to use and how to choose the ones that result in the best bandwidth. While in some modes NDI can automatically balance bandwidth across multiple NICs it is normally better for you to use NIC teaming at a machine configuration level which can result in much better performance than what is possible in software.

If this setting is configured incorrectly to specify NICs that might not exist, then NDI might fail to function correctly. Also please note that the operation of computer systems that are separately on entirely different networks with different IP address ranges is often not handled robustly by the operating system and NDI might not fully function in these configurations.

```
"rudp": {
  "send": {
    "enable": true
  },
  "recv": {
    "enable": true
  }
},
```

The following connections are available in NDI version 5 and allow the force enabling and disabling of the reliable UDP mode which is the default connection type on NDI version 5 and later. The full details of this connection type are described in the section for “Performance and Implementations” section of this manual.

There are separate settings for sending and receiving. Both sides need to allow this mode to be applied; sources and receivers have it enabled by default.

This is the default connection type and represents the preferred type for most network configurations and we recommend it’s use where possible.

```
"multicast": {
  "send": {
    "ttl": 1,
    "enable": false,
    "netmask": "255.255.0.0",
    "netprefix": "239.255.0.0"
  },
  "recv": {
    "enable": true
  }
```

These settings enable or disable the use of multicast for receiving. If you explicitly disable it on a machine then, even if the sender is configured for multicast, it will use unicast. When multicast receiving is enabled and a sender is available in the same local network, the receiver can negotiate for a multicast stream to be sent. If the sender is not on the same local network, this

```
},
```

negotiation does not occur (since it could lead to a multicast stream being sent but never able to arrive at the receiver). If you have a correctly configured network and can ensure a multicast stream can route reliably from a different network to the receiver's local network, you can specify the sender's subnet in the "subnets" setting to allow multicast negotiation to occur.

These settings pertain to multicast NDI setting on this machine. The first setting determines whether multicast sending is enabled or not. By default, multicast sending is disabled. Next is the IP address prefix and mask. In this example, multicast IP addresses will be chosen in the range 239.255.0.0 - 239.255.255.255. NDI will attempt to use different multicast addresses to ensure that the streams can be filtered efficiently by the network adapter. NDI senders need a range of multicast addresses available. The TTL value controls how many "hops" the multicast sending traffic will take, allowing it to move outside of the local network.

There are separate settings for sending and receiving. Both sides need to allow this mode to be applied; sources and receivers have it enabled by default.

We generally discourage the use of Multicast since configuration and ensuring that high performance is achieved is very difficult at a network level in most cases the default protocols (particularly reliable UDP) perform much better.

```
"tcp": {
  "send": {
    "enable": false
  },
  "recv": {
    "enable": false
  }
},
```

These settings enable or disable multi-TCP sending or receiving. If multi-TCP is disabled, then unicast UDP will be used. If unicast UDP is also disabled, then the base TCP connection will be used.

There are separate settings for sending and receiving. Both sides need to allow this mode to be applied; sources and receivers have it enabled by default.

Multi-TCP is not the default mode in NDI version 5 which sees better performance with Reliable UDP.

```
"unicast": {
  "send": {
    "enable": false
  },
  "recv": {
    "enable": false
```

These settings enable or disable unicast UDP sending or receiving. If unicast UDP is disabled, then the base TCP connection will be used.

Unicast settings determine whether UDP with forwards-

```
      }
    },
```

error correction is used for sending. While configurable, we recommended that this is enabled by default and not changed. Our experience has been that our UDP implementation handles poor networks and packet loss more robustly than TCP/IP, which can encounter timeout problems when acknowledgment packets are dropped (while rare, over a period of hours this can and does happen).

The UDP implementation also fully implements paced network sending with zero memory copy scatter-gather lists and jittered timing, to reduce the chance of packet loss on networks with many synchronized video streams. By default, 4Kb UDP packets are used, although jumbo packets do not need to be enabled on the network.

All versions of NDI fall to TCP/IP if a particular protocol is not supported by both sides. Again, note that that a sender implementation can simultaneously send internally in multiple modes, based on what receivers require.

There are separate UDP unicast settings for sending and receiving. Both sides need to allow this for UDP mode to be applied; sources and receivers have it enabled by default.

Unicast UDP is not the default mode in NDI version 5 which sees better performance with Reliable UDP.

```
"codec": {
      "shq": {
        "quality": 100,
        "mode": "auto"
      }
    },
  }
}
```

These settings are only available in the NDI Advanced SDK and allow you to over-ride the default codec quality settings of NDI. The “quality” setting is a percentage scale to apply to the bit-rate control, for instance a value of 200 would mean that NDI targets a bitrate that is double the NDI default. Be careful when specifying high bitrates because the CPU usage required for compression and decompression might increase and the strain on the network and the OS networking stack is correspondingly increased. Once the bitrate hits a maximum level for a particular media type (e.g., the codec q value become the maximum) then increasing it further might have no impact.

The “mode” allows you to force NDI into a particular color-mode. The default is “auto” which uses heuristics to best allocate bits between the luminance and

chroma fields. You may specify “4:2:2” or “4:2:0” here to force the codec into a particular chroma-subsampling mode. Please note that often forcing it into a particular mode will cause the codec to be less high quality than letting the codec choose the bit allocation that results in the best PSNR.

12 PLATFORM CONSIDERATIONS

Of course, all platforms are slightly different, and the location of configuration files and the settings can differ slightly between platforms. On all platforms, if there is an environment variable `NDI_CONFIG_DIR` set before initializing the SDK then we will load the ndi-config.v1.json from this folder when the library is used.

12.1 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. *Please note that the next major version of NDI will deprecate support for 32bit windows platforms.*

We have found that on some computer systems, if you install "WireShark" to monitor network traffic, a virtual device driver called "NPCap Loopback Driver" it installs can interfere with NDI, potentially causing it to fail to communicate. 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 it is installed on your system, we recommend that you go to your network settings and use the context menu on the adapter to disable it.

The NDI Tools bundle for Windows includes “Access Manager”, a user interface for configuring most of the settings outlined above. These settings are also stored in `C:\ProgramData\NDI\ndi-config.v1.json`.

12.2 WINDOWS UWP

Unfortunately, the Universal Windows Platform imposes significant restrictions that can negatively affect NDI, and about which you need to be aware. 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.

- Due to sandboxing, UWP applications cannot load external DLLs. This makes it unlikely that NDI|HX will work correctly.

- When you create a new UWP project you must ensure you have all 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)

12.3 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. Reliable UDP support requires macOS 10.14 or later which enabled IPv6 socket properties; NDI will work on earlier versions, but Reliable UDP will not be used.

Because of recent changes made within macOS in regard to signing of libraries, if you wish NDI|HX version 1 to work within your applications, you should locate the options in XCode under "Targets->Signing & Capabilities" and ensure that the option "Hardened Runtime -> Disable Library Validation" is checked.

In iOS 14 and XCode 12, a new setting was introduced to enable support for mDNS and Bonjour. Under "Bonjour Services", one should assign "Item 0" as being "_ndi._tcp." in order for NDI discovery to operate correctly.

The configuration settings are stored in `$HOME/.ndi/ndi-config.v1.json`.

12.4 ANDROID

Because Android handles discovery differently than other NDI platforms, some additional work is needed. The NDI library requires use of the "NsdManager" from Android and, unfortunately, there is no way for a third-party library to do this on its own. As long as an NDI sender, finder, or receiver is instantiated, an instance of the NsdManager will need to exist to ensure that Android's Network Service Discovery Manager is running and available to NDI.

This is normally done by adding the following code to the beginning of your long running activities:

```
private NsdManager m_nsdManager;
```

At some point before creating an NDI sender, finder, or receiver, instantiate the NsdManager:

```
m_nsdManager = (NsdManager)getSystemService(Context.NSD_SERVICE);
```

You will also need to ensure that your application has configured to have the correct privileges required for this functionality to operate.

12.5 IOS

iOS supports NDI finding, sending, and receiving.

In iOS 14 and XCode 12, a new setting was introduced to enable support for mDNS and Bonjour. Under "Bonjour Services", one should assign "Item 0" as being "_ndi._tcp." In order for NDI discovery operate correctly. It is also required to enable networking in the App sandbox settings.

The configuration settings are stored in `$HOME/.ndi/ndi-config.v1.json`.

12.6 LINUX

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.

The configuration settings are stored in `$HOME/.ndi/ndi-config.v1.json`.

Please take careful note on the comments under Linux in the “Reliable UDP” in the “Performance and Implementation” section.

13 NDI-SEND

A call to `NDIlib_send_create` will create an instance of the sender. This 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 using the current executable name, ensuring that there is a count that ensures sender names are unique (e.g. “My Application”, “My Application 2”, etc.)

Supported Parameters	
p_ndi_name (const char*)	This is the name of the NDI source to create. It is a `NULL`-terminated UTF-8 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”, 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 the 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 they are submitted at. The same is true for audio. In general, if you are submitting video and audio off a single thread, you should only clock one of them (video is probably the better choice 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 how works is that, when you submit a frame, it will keep track of the time the next frame would be required at. If you 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 framerate 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 send_create;
```

```
send_create.p_ndi_name = "My Video";
send_create.p_groups = NULL;
send_create.clock_video = true;
send_create.clock_audio = true;

NDIlib_send_instance_t pSend = NDIlib_send_create(&send_create);
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 metadata frames to the device – at any time, off any thread, and in any order.

There are no reasonable restrictions on video, audio or metadata 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!)
uint8_t* p_frame = (uint8_t*)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 = 0;
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 48 kHz

// 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 = 0;
audio_frame.p_data = p_frame;
audio_frame.channel_stride_in_bytes = sizeof(float)*1920;
audio_frame.p_metadata = NULL; // No metadata on this example!

// Submit the buffer
NDIlib_send_send_audio_v3(pSend, &audio_frame);

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

Because many applications provide interleaved 16-bit audio, the NDI library includes utility functions to convert PCM 16-bit 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. (Please refer 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 as this is easily understood by referring to the audio and video examples.)

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 refer you to the way 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 to efficiently wait for messages on a thread. The basic process is outlined below:

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

    // Free the metadata message
    NDIlib_recv_free_metadata(pSend, &meta_data);
}
```

Connection metadata, as specified in the NDI-Recv section of this documentation, is an important category of metadata 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 and can include hints as to the capabilities the receiver might offer. Examples include the resolution and framerate 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 metadata 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)
{
    // 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. To add a metadata element, one can call `NDIlib_send_add_connection_metadata`. To clear all 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 metadata registration that allows people to know what we are.
NDIlib_metadata_frame_t NDI_product_type;
NDI_product_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(pSend, &NDI_product_type);
```

Because NDI assumes that all senders must have a unique name and 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 lifetime of the returned value is until the sender instance is destroyed.

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

Using this in conjunction with a clocked video output results in a very efficient rendering loop where you do not need to use separate threads for timing or for frame submission. For example, 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(pSend, &frame_data);
}

NDIlib_send_send_video_v2_async(pSend, NULL); // Sync here
```

Note: User error involving asynchronous sending is most common SDK 'bug report'. It is very important to understand that a call to `NDIlib_send_send_video_v2_async` starts processing, then sending the video frame asynchronously with the calling application. If you call this and then free the pointer, your application will most likely crash in an NDI thread – because the SDK is still 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 likely exhibit tearing or other glitches, since you are writing to the buffer while the SDK is still compressing data it previously held. 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 being asynchronous to the others.

Note: If you are using the Advanced SDK, it is possible to assign a completion handler for asynchronous frame sending that more explicitly allows you to track buffer ownership with asynchronous sending.

13.2 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 100 ns 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. This keeping your streams exactly in sync, as long as the frames you are sending do does 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. And if you have NTP installed on your local network, 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 any meaningful deviation from the timecode you specified over time.

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 no sample looks sufficiently 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.

13.3 FAILSAFE

Failsafe is a capability of any NDI sender. If you specify a failsafe source on an NDI sender and the sender fails for any reason (even the machine failing completely), any receivers viewing that sender will automatically switch over to the failsafe sender. If the failed source comes back online, receivers will switch back to that source.

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

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

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

13.4 CAPABILITIES

An NDI capabilities metadata message can be submitted to the NDI sender for communicating certain functionality that the downstream NDI receivers should know about upon connecting. For example, if you are providing PTZ type functionality, letting the NDI receiver know this would done through this type of metadata message. The following is an example of the NDI capabilities message:

```
<ndi_capabilities web_control="http://ndi.tv/" ntk_ptz="true" ntk_exposure_v2="true" />
```

You would submit this message to the NDI sender for communication to current and future NDI receivers as follows:

```
NDIlib_metadata_frame_t NDI_capabilities;
NDI_capabilities.p_data = "<ndi_capabilities web_control=\"http://ndi.tv/\" "
                          "                  ntk_ptz=\"true\" "
                          "                  ntk_exposure_v2=\"true\" />";
NDIlib_send_add_connection_metadata(pNDI_send, &NDI_capabilities_type);
```

Below is a table of XML attributes that can be used in this capabilities message:

Supported Attributes	
web_control	The URL to the local device webpage. If `%IP%` is present in the value, it will be replaced with the local IP of the NIC in which the NDI receiver is connected to.
ntk_ptz	Signifies that this NDI sender is capable of processing PTZ commands sent from the NDI receiver. The NDI receiver will only assume the NDI sender can support PTZ commands if this attribute is received and set to the value "true".
ntk_pan_tilt	The NDI sender supports pan and tilt control.
ntk_zoom	The NDI sender supports zoom control.
ntk_iris	The NDI sender supports iris control.
ntk_white_balance	The NDI sender supports white balance control.
ntk_exposure	The NDI sender supports exposure control.
ntk_exposure_v2	The NDI sender supports detailed control over exposure such as iris, gain, and shutter speed.
ntk_focus	The NDI sender supports manual focus control.
ntk_autofocus	The NDI sender supports setting auto focus.
ntk_preset_speed	The NDI sender has preset speed support.

13.5 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 closing certain networking operations when the app is placed in the background, then restarted when put in 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.

14 NDI-FIND

This is provided to locate sources available on the network and is normally used in conjunction with NDI-Receive. Internally, it uses a cross-process P2P mDNS implementation to locate sources on the network. (It commonly takes a few seconds to locate all 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. Be aware that some network routers might block mDNS traffic between network segments.

Creating the find instance is very similar to the other APIs – one fills out a `NDIlib_find_create_t` structure to describe the device that is needed. It is possible to specify a `NULL` creation parameter in which case default parameters are used.

If you wish to specify the parameters manually, then the member values are as follows:

Supported Values	
show_local_sources (bool)	This flag tells the finder whether it should locate and report NDI send sources that are running on the current local machine.
p_groups (const char*)	This parameter specifies groups for which this NDI finder will report sources. A full description of this parameter and what a `NULL` default value means is provided in the description of the NDI-Send SDK.
p_extra_ips (const char*)	This parameter will specify a comma separated list of IP addresses that will be queried for NDI sources and added to the list reported by NDI find. These IP addresses need not be on the local network and can be in any IP visible range. NDI find will be able to find and report any number of NDI sources running on remote machines and will correctly observe them coming online and going offline.

Once you have a handle to the NDI find instance, you can recover the list of current sources by calling `NDIlib_find_get_current_sources` at any time. This will *immediately* return with the current list of located sources.

The pointer returned by `NDIlib_find_get_current_sources` is owned by the finder instance, so there is no reason to free it. It will be retained until the next call to `NDIlib_find_get_current_sources`, or until the `NDIlib_find_destroy` function is destroyed.

You can call `NDIlib_find_wait_for_sources` to wait until the set of network sources has been changed; 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, calls `NDIlib_find_get_current_sources` to get the current list of sources:

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

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

while (true) // You would not loop forever of course !
{
    // Wait up till 5 seconds to check for new sources to be added or removed
    if (!NDIlib_find_wait_for_sources(pFind, 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(pFind, &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 the sources on the network; these will be added (or removed) as additional or new sources are discovered. It commonly takes a few seconds to discover all sources on a network.

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

15 NDI-RECV

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

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

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 `NULL` source if you wish to create a receiver that you desire to connect at a later point with `NDIlib_recv_connect`.
p_ndi_name	This is a name that is used for the receiver and will be used in future versions of the SDK to allow discovery of both senders and receivers on the network. This can be specified as `NULL` and a unique name based on the application executable name will be used.
color_format	This parameter determines what color formats you are passed when a frame is received. In general, there are two color formats used in any scenario: one that exists when the source has an alpha channel, and another 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 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 in this mode. On most platforms this will return an 8-bit UYVY video buffer when there is no alpha channel, and an 8-bit 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 16-bpp Y+Cb,Cr (P216 FourCC) buffer when the underlying codec is native NDI, or a 8-bpp UYVY buffer when the native codec is an 8-bit codec like H.264. When there is alpha channel this will normally return a 16-bpp 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 other alternatives may be available in 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 significantly reduced bandwidth.

allow_video_fields	If your application does not like receiving fielded video data you can specify `false` to this value, and all video received will be de-interlaced before it is passed to you. The default value should be considered `true` for most applications. The implied value is `true` when `color_format` is `NDIlib_recv_color_format_fastest`.
p_ndi_name	This is the name of the NDI receiver to create. It is a `NULL`-terminated UTF-8 string. Give your receiver a meaningful, descriptive, and unique name. This will be the name of the NDI receiver on the network. For instance, if your network machine name is called "MyMachine" and you specify this parameter as "Video Viewer", then the NDI receiver on the network would be "MyMachine (Video Viewer)".

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

If you create a receiver with `NULL` as the settings, or if you wish to change the remote source that you are connected to, you may call `NDIlib_recv_connect` at any time with a `NDIlib_source_t` pointer. If the source pointer is `NULL` 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 metadata frames by calling `NDIlib_recv_capture_v3`. 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 `NULL`. It can safely be called across many threads at once, allowing you to have one thread receiving video while another receives audio.

The `NDIlib_recv_capture_v3` function takes a timeout value specified in milliseconds. If a frame is available when you call `NDIlib_recv_capture_v3`, 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 returns 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_v3(pRecv, &video_frame, &audio_frame, &metadata_frame, 1000))
{
    // We received video.
    case NDIlib_frame_type_video:
        // Process video here
        // Free the video.
        NDIlib_recv_free_video_v2(pRecv, &video_frame);
        break;

    // We received audio.
    case NDIlib_frame_type_audio:
        // Process audio here
```

```
            // Free the audio.
            NDIlib_recv_free_audio_v3(pRecv, &audio_frame);
            break;

        // We received a metadata packet
        case NDIlib_frame_type_metadata:
            // Do what you want with the metadata 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 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.

`NDIlib_recv_capture_v3` 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 metadata frames have been dropped, you can call `NDIlib_recv_get_performance`, which will supply the total frame count and 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 metadata (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 metadata is an important concept that allows you to "register" certain metadata messages so that each time a new connection is established the up-stream source (normally an NDI Send user) receives those strings. Note that there are many reasons that connections might be lost and established at run time.

For instance, if an NDI-Sender goes offline the connection is lost; if it comes back online at a later time, the connection would be re-established, and the connection metadata would be resent. Some standard connection strings are specified for connection metadata, as outlined in the next section.

Connection metadata 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 metadata registration that allows people to know what we are.
NDIlib_metadata_frame_t NDI_product_type;
NDI_product_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(pRecv, &NDI_product_type);
```

Note: When using the Advanced SDK, it is possible to assign custom memory allocators for receiving that will allow you to provide user-controlled buffers that are decompressed into. In some cases, this might improve performance or allow you to receive frames into GPU accessible buffers.

15.1 RECEIVER USER INTERFACES

A sender might provide an interface that allows configuration. For instance, an 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 Advanced SDK 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 `NULL` if there is no current URL associated with the sender in question. Because connections might take a few seconds, this string might not be available immediately after having called connect. To avoid the need to poll this setting, note that `NDIlib_recv_capture_v3` returns 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(pRecv);
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(pRecv, 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.

For sources indicating they support the ability to be configured, NewTek's NDI Studio Monitor application includes this capability 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.

15.2 RECEIVER PTZ CONTROL

NDI standardizes the control of PTZ cameras. An NDI receiver will automatically sense whether the device it is connected to 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.

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 using the following call:

```
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 before the source indicates that it supports PTZ control. To avoid the need to poll this setting, note that `NDIlib_recv_capture_v3` returns a value of `NDIlib_frame_type_status_change` when this setting is known (or when it has changed).

15.2.1 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 that the host application control the basic messages listed below.

15.2.1.1 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.

15.2.1.2 ZOOM SPEED

```
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.

15.2.1.3 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.

15.2.1.4 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.

15.2.1.5 FOCUS

Focus on cameras can either be in auto-focus mode or in manual focus mode. The following 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.

15.2.1.6 WHITE BALANCE

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

```
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.

15.2.1.7 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_v2(NDIlib_recv_instance_t p_instance,
                                        const float iris,
                                        const float gain,
                                        const float shutter_speed);
```

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

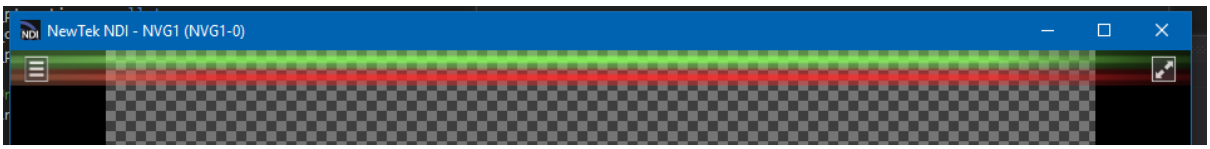
15.3 RECEIVERS AND TALLY MESSAGES

Video receivers can specify whether the source is visible on a video switcher's program or preview row. This is communicated up-stream to the source's sender, which then indicates its state (see the section on the sender SDK within this document). The sender takes its state and echoes it to all receivers as a metadata 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, consider a sender named "My Source A" sending to two destinations, "Switcher" and "Multi-viewer". When "Switcher" places "My Source A" on program out, a tally message is sent from "Switcher" to "My Source A". Thus, the source '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 display an indicator when the source monitored has its tally state set.



15.4 FRAME SYNCHRONIZATION

When using video, it is important to realize that different clocks are often used by 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 rated to operate at 48000 Hz. However, it might well actually run at 48001 Hz, or perhaps 47998 Hz.

Similar variances also affect receivers. While the differences appear miniscule, they can accumulate to cause significant audio sync 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 of the diverse scenarios that can arise, 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, NDI's 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 a screen or multiviewer	Yes – you want the clock to be synced with vertical refresh. On a multiviewer 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, metadata, etc. Remember, it 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 use 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 metadata are inserted into the best possible audio samples.

Also, if you specify the desired sample-rate as zero it 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
    int sample_rate, // Your desired sample rate. 0 for "use source".
    int no_channels, // Your desired channel count. 0 for "use source".
    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 to provide 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
    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);
```

16 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 to "Video Routing 1", the data they receive will actually be from "Video Source 1".

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

You create a video routing source using:

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

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

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

and:

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

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

```
void NDIlib_routing_destroy(NDIlib_routing_instance_t p_instance);
```

17 COMMAND LINE TOOLS

17.1 RECORDING

A, full cross-platform native NDI recording is provided in the SDK. This is provided as a command line application in order to allow it to be integrated into both end-user applications and scripted environments. 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 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 efficiency of the system network and disk device drivers.
- **All sources are synchronized**. The recorder will time-base correct all recordings to lock to the current system clock. This is designed so that, if you are recording a large number of NDI sources, the resulting files are entirely synchronized with each other. Because the files are written with timecode, they may then be used in a nonlinear editor without any additional work required for multi-angle or multi-source synchronization.
- Still better, if you lock the clock between multiple computers systems using NTP, recordings done independently on all computer systems will always be automatically 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.** Using asynchronous block-based disk writing without any video compression in most cases means that the number of streams written to disk is largely limited only by 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 wanting recording capabilities, we realized that providing a reference implementation that handles a lot of the edge-cases and problems of recording would be hugely beneficial. And allowing all sources to be synchronized makes NDI a fundamentally more powerful and useful tool for video in all cases.
- The implementation provided is cross-platform, and may be used under the SDK license in commercial and free applications. Note that audio is recorded in floating-point and so is never subject to audio clipping at record time.

[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.

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

17.1.1 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 in order to determine the format 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 already exists a number will be appended to it 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.
-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, framerate, aspect ratio) the existing file is chopped and new one started 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 it will 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 its launching parent process; if that should exit it will correctly close the file and exit.

17.1.2 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 and 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 1.2 dB 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.

17.1.3 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)2020 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 specifying the filename for the recording, and provide you with the framerate.

It then gives you the timecode 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. Timecodes are specified as UTC time since the Unix Epoch (1/1/1970 00:00) with 100 ns precision.

17.1.4 ERROR HANDLING

A number of different events can cause recording errors. The most common is when the drive system that you are recording to is too slow 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 much as 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. Thus, 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, the recorder will start attempting to write a new file. This process ensures that each file always has all frames without drops, but if data needed to be dropped because of insufficient disk performance, that data will be missing between files.

17.2 NDI DISCOVERY SERVICE

The NDI discovery service is designed to allow you to replace the automatic discovery NDI uses with a server that operates as a centralized registry of NDI sources.

This can be very helpful for installations where you wish to avoid having significant mDNS traffic for a large number of sources, or in which multicast is not possible[2] or desirable. When using the discovery server, NDI is able to operate entirely in unicast mode and thus in almost any installation.

The discovery server supports all NDI functionality including NDI groups.

17.2.1 SERVER

Using a discovery server 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 accepts incoming connections with senders, finders, and receivers, and coordinates amongst them all to ensure they are all visible to each other.

If you wish to bind the discovery server to a single NIC, then you can run it with a command line that specifies the NIC to be used. For instance:

```
"NDI Discovery Service.exe" -bind 196.168.1.100
```

Note: If you are installing this on a separate machine from the SDK you should ensure that the Visual Studio 2019 C runtime is installed on that machine, and that the NDI licensing requirements are met.

32-bit and 64-bit versions of the discovery service are available, although the 64-bit version is recommended. The server will use very little CPU usage although, when there are a very large number of source and connections, it might require RAM and some network traffic between all sources to coordinate source lists.

Hint: It is, of course, recommended that you have a static IP address so that any clients configured to access it will not lose connections if the IP is dynamically re-assigned.

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

17.2.2 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* so as 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; these sources will *only* be visible to other finders and receivers that are configured to use the discovery server.

17.2.3 CONFIGURATION

In order to configure the discovery server for NDI clients, you may use Access Manager (included in the NDI Tools bundle) to enter the IP address of the discovery server machine.

To configure the discovery server for NDI clients, you may use Access Manager (included in the NDI Tools bundle) to enter the IP address of the discovery server machine.

It is possible to run the Discovery server with a command line option that specifies which NIC it is operating from with:

```
DiscoveryServer.exe -bind 192.168.1.100
```

This will ask the discovery server to only advertise on the IP address specified. Likewise, it is possible to specify a port number that will be used for the discovery server using:

```
DiscoveryServer.exe -port 5400
```

This allows you to work on a non-default port number or run multiple discovery servers for multiple groups of sources on a single machine. If a port of 0 is specified, then a port number is selected by the operating system and will be displayed at run-time.

17.2.4 REDUNDANCY AND MULTIPLE SERVERS

Within NDI version 5 there is full support for redundant NDI discovery servers. When one configures a discovery server it is possible to specify a comma delimited list of servers (e.g. “192.168.10.10, 192.168.10.12”) and then they will all be used simultaneously. If one of these servers then goes down then as long as one remains active then all sources will remain visible at all times; as long as at least one server remains active then no matter what the others do then all sources can be seen.

This multiple server capability can also be used to ensure entirely separate servers to allow sources to be broken into separate groups which can serve many workflow or security needs.

17.3 NDI BENCHMARK

In order to help gauge your machine performance for NDI, a tool is provided that will initiate one NDI stream per core on your system and measure how many 1080p streams can be encoded in real time. Note that this number reflects the best case performance and is designed to exclude any impact of networking and only gauge the system CPU performance.

This can be used to compare performance across machines and because NDI is highly optimized on all platforms it is a good measure of the total CPU performance that is possible when all reasonable opportunity is taken to achieve high performance on a typical system. For instance, on Windows NDI will use all extended vector instructions (SSSE3 and up, including VEX instructions) while on ARM it will use NEON instructions when possible.

18 FRAME TYPES

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

18.1 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_video_type_e)	This is the pixel format for this buffer. The supported formats are 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 8-bit and 0-65536 range when running 16-bit.)

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

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 = (float)frame_rate_N / (float)frame_rate_D` Some examples of common framerates are presented in the table below.

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

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 it is interpreted as xres/yres, or square pixel; 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 field 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:

18.1.1.1 NTSC 486 LINES

The best way to handle this format is simply to offset the image vertically 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.

18.1.1.2 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, 64-bit signed integer)	This is the timecode of this frame in 100 ns 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 metadata stream that should be in UTF-8 formatted XML and NULL-terminated. It is sent and received with the frame.
timestamp (int64_t, 64-bit signed integer)	This is a per-frame timestamp filled in by the NDI SDK using a high precision clock. It represents the time (in 100 ns 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 ~1 µs 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).

18.2 AUDIO FRAMES (NDILIB_AUDIO_FRAME_V3_T)

NDI Audio is passed to the SDK in floating-point and has a dynamic range without practical limits (without clipping). To define how floating-point values map into real-world audio levels, a sinewave 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 +4 dBU, corresponding to a nominal level of 1.228 V 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	-∞	-20 dB	-16 dB	+0 dB	+4 dB	+24 dB
dBVU	-∞	-24 dB	-20 dB	-4 dB	+0 dB	+20 dB
SMPTE dBFS	-∞	-44 dB	-40 dB	-24 dB	-20 dB	+0 dB

If you want a simple 'recipe' that matches SDI audio levels based on the SMPTE audio standard, you will want to have 20 dB of headroom above the SMPTE reference level at +4 dBu, which is at +0 dBVU, 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_16bit = 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	-∞	-20 dB	-16 dB	+0 dB	+4 dB	+18 dB
dBVU	-∞	-24 dB	-20 dB	-4 dB	+0 dB	+14 dB
EBU dBFS	-∞	-38 dB	-34 dB	-18 dB	-14 dB	+0 dB

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

```
int ebu_sample_16bit = max(-32768, min(32767, (int)(6540.52f*ebu_sample_fp)));
```

Because many applications provide interleaved 16-bit audio, the NDI library includes utility functions that will convert in and out of floating-point formats from PCM 16-bit formats.

There is also a utility function for sending signed 16-bit audio using *NDIlib_util_send_send_audio_interleaved_16s*. Please refer to the example projects, and 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, 64-bit signed integer)	This is the timecode of this frame in 100 ns 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 100 ns 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.
FourCC (NDIlib_FourCC_audio_type_e)	This is the sample format for this buffer. There is currently one supported format: NDIlib_FourCC_type_FLTP. This format stands for floating-point audio.
p_data (uint8_t*)	If FourCC is NDIlib_FourCC_type_FLTP, then 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 metadata stream that should be in UTF-8 formatted XML and `NULL`-terminated. It is sent and received with the frame.
timestamp (int64_t, 64-bit signed integer)	This is a per-frame timestamp filled in by the NDI SDK using a high precision clock. It represents the time (in 100 ns 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 ~1 µs 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.

18.3 METADATA FRAMES (NDILIB_METADATA_FRAME_T)

Meta data is specified as `NULL`-terminated, UTF-8 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 metadata message in bytes. It includes the `NULL`-terminating character. If this is zero, then the length will be derived from the string length automatically.
p_data (char*)	This is the XML message data.
timecode (int64_t, 64-bit signed integer)	This is the timecode of this frame in 100 ns 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 namespace 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 metadata flags to be standardized, please contact us.

19 WINDOWS 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/NDIRedistV5. 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 16-bit, and video is supported in UYVY and BGRA.

The filter can be added to a graph and will respond to the IFileSourceFilter interface. This 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 must 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
```

20 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.

21 SUPPORT

Like other areas of the NDI® SDK, if you have any problems, please sign up on our NDI support hub at https://www.ndi.tv/ndiplusdevhub and we will do our best to support you.