NDI®

ADVANCED SDK VERSION 5.5

8/9/2022

Table of Contents

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. In particular, the NDI Advanced SDK has been prepared with the needs of device manufacturers who wish to provide hardware assisted encoding or decoding in mind (see Section 23, NDI Advanced SDK).

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 ADVANCED SDK DEVICE SUPPORT

As mentioned earlier, several options are available to provide NDI® support to Advanced SDK systems or hardware devices. The NDI Advanced SDK includes details to allow NDI to be compressed on smaller FPGA designs or, from version 4, to leverage existing H.265, H.264 and AAC encoders already on a device by simply updating its firmware to support specific requirements (this approach lets you quickly and easily add NDI support to *existing* products).

5 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.

5.1 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.

5.2 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.

5.3 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`.

5.4 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.

5.5 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.

5.5.1 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.

5.5.2 NDI-FIND

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

5.5.3 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.

5.6 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.

5.7 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.

6 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.

7 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.

7.1 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.

7.2 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.

7.3 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(...)`

8 PERFORMANCE AND IMPLEMENTATION

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

8.1 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.

8.2 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

8.6.1 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.

9 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.

10 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.

11 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

12 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,
```

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

```
    "netmask": "255.255.0.0",
    "netprefix": "239.255.0.0"
  },
  "recv": {
    "enable": true
  }
},
```

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.

13 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.

13.1 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`.

13.2 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)

13.3 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`.

13.4 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.

13.5 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`.

13.6 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.

14 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.

14.1 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.

14.2 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.

14.3 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.

14.4 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.

14.5 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.

15 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`.

16 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.

16.1 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);
```

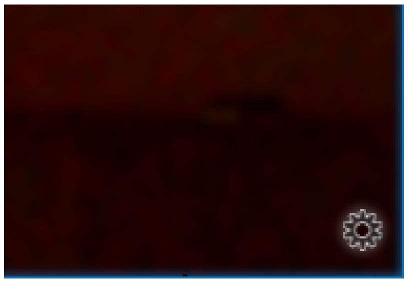
This will return a string representing the URL, or `NULL` if there is no current URL associated with the sender in question. Because connections might take a few seconds, this string might not be available immediately after having called connect. To avoid the need to poll this setting, note that `NDIlib_recv_capture_v3` 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.
}
```

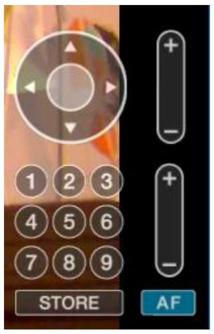
You can then store this URL and provide it to an end user as the options for that device. For instance, a PTZ camera or an NDI conversion box might allow its settings to be configured using a hosted web interface.

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.

16.2 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).

16.2.1 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.

16.2.1.1 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.

16.2.1.2 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.

16.2.1.3 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.

16.2.1.4 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.

16.2.1.5 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.

16.2.1.6 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.

16.2.1.7 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].

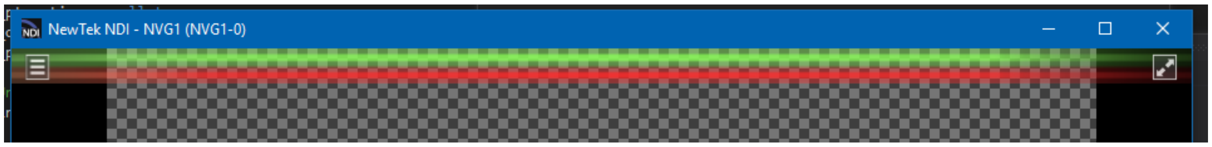
16.3 RECEIVERS AND TALLY MESSAGES

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.



16.4 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);
```

17 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);
```

18 COMMAND LINE TOOLS

18.1 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.

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`.

18.1.1 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.

[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.

-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.

18.1.2 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.

18.1.3 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.

18.1.4 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.

18.2 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.

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

18.2.1 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.

18.2.2 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.

18.2.3 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.

18.2.4 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.

18.3 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.

19 FRAME TYPES

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

19.1 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:

19.1.1.1 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.

19.1.1.2 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).

19.2 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.

19.3 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.

20 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
```

21 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.

22 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.

23 NDI ADVANCED SDK

23.1 OVERVIEW

This section of the NDI Advanced SDK is designed for use by device manufacturers who wish to provide hardware assisted encoding or decoding. It follows the same API as the NDI SDK so that any experience, sample code, documentation from one can be applied to the other.

Importantly, this part of the NDI Advanced SDK provides direct access to the video data in compressed form so that it can be sent and received directly. Thus, it is also likely that this SDK can be used for other tasks that require or benefit from interacting directly with the compressed video data for sending and receiving.

There are currently two primary uses of the NDI Advanced SDK.

1. You can use NDI compression (sometimes called "SpeedHQ"), a high performance and high quality I-Frame video codec. An FPGA compressor for native NDI is provided within this SDK.

   If you are using SpeedHQ compression, you are likely also using a (SoC) device with an ARM core and an FPGA that can be used for real-time compression. The NDI Advanced SDK provides NDI Encode and Decode IP cores for Xilinx and Altera FPGAs, along with example FPGA projects and reference C++ applications including full source code.

   To help get you started, prebuilt bootable uSD drive images for several standard Development kits are provided. The FPGA core supplied will easily encode 4K video in real-time on relatively modest FPGA designs, assuming the device has sufficient memory bandwidth (multiple banks of RAM are recommended); on latest generation SoC systems it can easily encode one or more streams of 8K if sufficient network bandwidth is provided.

2. You may use H.264 or H.265 for video and AAC audio with this SDK if you are developing using devices that already have hardware compressors for these available.

Because many Advanced SDK systems require custom tool chains, NewTek can provide a compiled version of the SDK expressly built for your specific system. Please email sdk@ndi.tv if you have such requirements.

23.2 CONFIGURATION FILES

All default settings for NDI® are controlled through a configuration file. This file is located at `$HOME/.ndi/ndi-config.v1.json`. It contains settings for multicast and unicast sending as well as vendor information. Your vendor ID is specified here and must be registered with NewTek for compressed data pass-through to be enabled.

The configuration file will automatically be loaded off disk by default and these settings used. If you wish to work this way, you can simply restart your application when configuration settings on the device are changed, and the settings will be updated.

Alternatively, the NDI creation functions can be passed in a string representation of this configuration file, allowing you to change it for senders and receivers by simply recreating them at run time.

There is a setting for `p_config_json` that may be set in creation of devices to permit an in-memory version of the configuration file to be passed in. An example showing how to set p_config_json, if that is your preferred method of specifying the configuration, follows below.

```
const char *p_config_json ="{"
    "\"ndi\": {"
```

```
                "\"vendor\": {"
                    "\"name\": \"CoolCo, inc.\","
                    "\"id\": \"00000000-0000-0000-0000-000000000000\""
                "}"
            "}"
        "}";
```

The ability to specify per connection settings for any sender, finder and receiver is a very powerful ability since it allows every connection to be completely customized per use, allowing for instance different NICs to be used for different connection types, multicast to be manually specified by connection, different groups and much more.

The full details of the configuration file are provided in the manual under the section on *Performance and Implementation Details*.

23.3 NDI SDK REVIEW

This section provides a brief introduction to using the NDI® SDK and demonstrates how to create and use a sender and receiver. While this is likely enough to get started developing applications, there are many additional examples and documentation in the SDK.

23.3.1 SENDING

NDI senders are created in exactly same way that they would be in the NDI SDK, using `NDIlib_send_create` or the newer function available to the Advanced SDK, `NDIlib_send_create_v2`.

Note: It is strongly recommended that your device allow its name to be configured, so that individual devices can be identified on the network.

You are also able to specify that your device exists within different NDI groups should you desire.

An example of creating a sender might be:

```
// Setup the structure describing the sender
NDIlib_send_create_t send_create;
send_create.clock_audio = false; // Your audio is probably clocked by your own hardware.
send_create.clock_video = false; // Your video is probably clocked by your own hardware.
send_create.p_ndi_name = "Your name"; // Often configured in your web page.
send_create.p_groups = NULL; // You can allow this to be configured in your web page.

const char *p_config_json = NULL; // You can override the default json file settings
NDIlib_send_instance_t pSend = NDIlib_send_create_v2(&send_create, p_config_json);
```

It is crucial that you provide an XML identification for all hardware devices. This should include your vendor name, model number, serial number and firmware version.

For example, the following XML identification would work, although it should be filled in with the correct settings for your device (serial numbers should be unique to each device manufactured):

```
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.\" "
                          "             version=\"1.000.000\" "
                          "             session=\"default\" "
                          "             model_name=\"S1\" "
                          "             serial=\"ABCDEFG\" />";
NDIlib_send_add_connection_metadata(pNDI_send, &NDI_product_type);
```

You are now going to be able to pass compressed frames directly to the SDK. The following is a very basic example of how this might be configured.

```
NDIlib_video_frame_v2_t video_frame;
video_frame.xres = 1920;
video_frame.yres = 1080;
video_frame.frame_format_type = NDIlib_frame_format_type_progressive;
video_frame.FourCC = (NDIlib_frame_format_type_e)NDIlib_FourCC_type_SHQ2_highest_bandwidth;
video_frame.picture_aspect_ratio = 16.0f/9.0f;
video_frame.frame_rate_N = 60000;
video_frame.frame_rate_D = 1001;
video_frame.timecode = /* A timestamp from your hardware at 100 ns clock rate */;
video_frame.p_data = /* Your compressed data */;
video_frame.line_stride_in_bytes = /* The size of your compressed data in bytes */;

NDIlib_send_send_video_v2(pSend, &video_frame);
```

It is important to always submit both a program stream and a preview stream to the SDK. The program stream should be the full resolution video stream. The preview stream should always be progressive, have its longest dimension as 640 pixels, and a framerate that does not exceed 45 Hz.

- It is considered acceptable to simply drop every second frame to achieve the correct preview framerate when needed.
- It is also acceptable to always scale down by an integer scaling factor to be close to the correct resolution. (Scaling quality is not defined, although high quality scaling is preferred.)

Examples of the preview stream are listed below:

Main Video Format	Preview Video Format
1920x1080, 16:9, 59.94 Hz	640x360, 16:9, 29.97 Hz
1920x1080, 16:9, 29.97 Hz	640x360, 16:9, 29.97 Hz
1080x1920, 9:16, 50 Hz	360x640, 9:16, 25 Hz

In order to submit a preview resolution frame, one would use the following header (compare with previous example).

```
NDIlib_video_frame_v2_t video_frame;
video_frame.xres = 640;
video_frame.yres = 360;
video_frame.frame_format_type = NDIlib_frame_format_type_progressive;
video_frame.FourCC = (NDIlib_frame_format_type_e)NDIlib_FourCC_type_SHQ2_lowest_bandwidth;
video_frame.picture_aspect_ratio = 16.0f/9.0f;
video_frame.frame_rate_N = 30000;
video_frame.frame_rate_D = 1001;
video_frame.timecode = /* A timestamp from your hardware at 100 ns clock rate */;
video_frame.p_data = /* Your compressed data */;
NDIlib_send_send_video_v2(pSend, &video_frame);
```

It is common on Advanced SDK devices that you wish to have the SDK send frames without needing to wait for them to be complete. The asynchronous operations supported by the NDI SDK have been fully extended to the NDI Advanced SDK, and we recommend that these are used for best hardware performance.

The Advanced SDK has been designed to perform zero memory copy sending of frames over the network when using async operations. The SDK will assume that it can access each async buffer until either a) the next time that `NDIlib_send_send_video_async_v2` is called or b) the sender is closed.

Sending and receiving low and high bandwidth frames are entirely asynchronous with each-other. It is also possible to send main, preview, and audio streams from separate threads, should this be beneficial.

For instance, the following might be used as a send loop:

```
NDIlib_video_frame_v2_t frame_main, frame_prvw;

while (true)
{
    // Get the next frames to send
    NDIlib_video_frame_v2_t new_frame_main = get_frame_main();
    NDIlib_video_frame_v2_t new_frame_prvw = get_frame_prvw();

    // Send the frames
    NDIlib_send_send_video_async_v2(pSend, &new_frame_main);
    NDIlib_send_send_video_async_v2(pSend, &new_frame_prvw);

    // The previous frames are now guaranteed to no longer be needed by the SDK
    release_frame_main(&frame_main);
    release_frame_main(&frame_prvw);

    // These are now the next frames to use
    frame_main = new_frame_main;
    frame_prvw = new_frame_prvw;
}
```

Sending audio is nearly identical to sending video. The following example shows how to submit an audio 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.FourCC = /* Fill in with the correct value */;
audio_frame.p_data =  /* Fill in with the correct value */;
audio_frame.channel_stride_in_bytes = /* Fill in with the correct value */;
audio_frame.timecode = /* Fill in with a 100 ns timestamp from your device */;

NDIlib_send_send_audio_v3(pSend, &audio_frame);
```

It is crucial to ensure that your audio is provided to the SDK at the correct audio level, with a +4 dBU sinewave corresponding to a floating-point signal from -1.0 to +1.0. These levels can easily be debugged using the NDI Studio Monitor application.

The video bitrate should be controlled by your compressor by varying the Q level. You may determine this by calling the following function, which will return the size of the expected frame in bytes.

```
int NDIlib_send_get_target_frame_size(NDIlib_send_instance_t p_instance,
                                      const NDIlib_video_frame_v2_t* p_video_data);
```

There are many possible mechanisms available to perform bitrate control, and it is expected that you are within 10% of the returned value. (It is understood that it is often impossible to adapt the Q value for the frame being compressed and that the update will often occur on the subsequent frame.)

The examples above show how video may be sent using NDIlib_send_send_video_v2. By default, the SDK will not copy the memory buffers of video frames that are passed in (other than implicit copies required by your operating system for network sending events). There may be times when the video frame is composed of multiple pieces instead of a single contiguous block of memory. For the best performance, you can use one of the two scatter-gather sending functions provided to handle frames that are broken up into many pieces, either `NDIlib_send_send_video_scatter`, or `NDIlib_send_send_video_scatter_async`.

The `NDIlib_send_send_video_scatter_async` function will allow you to schedule entirely asynchronous sends. For this purpose, please review the comments above on buffer ownership and lifetime, which are the single most common problems reported due to incorrect SDK usage.

To work with high performance on high latency connections we very strongly recommend that you implement asynchronous sending completions, which are outlined in the next two sections. These allow data to be sent with NDI that does not require any memory-copies to send onto the network, while also allowing enough frames to be “in flight” that one can achieve high performance even on high latency networks.

Sending video is currently supported in 4:2:2:4, 4:2:2, and 4:2:0 formats (4:4:4:4 will be added in a future version; this is already internally supported by the software SDK). It has been our experience that that 4:2:0 yields better video quality at resolutions above 1920x1080, since more bits are assigned in the luminance.

23.3.1.1 ASYNCHRONOUS SENDING COMPLETIONS

When using asynchronous sending, the default way that it works is that you provide a buffer to the call and ownership of that buffer is held by the SDK until the return of the next asynchronous sending call. While this allows fully asynchronous sending, a problem that can occur is that if multiple receivers are connected to a source and one of those receivers is not responsive (either because it has become ungracefully disconnected or because it’s connection is slow) then sending a new frame via an asynchronous send might lock until sending to that network receiver is complete; causing potential video delays or freezes.

It is possible to assign an asynchronous completion handler for an NDI sender. When this is assigned, each call to an asynchronous sending function like `NDIlib_send_send_video_async_v2` and `NDIlib_send_send_video_scatter_async` will always return immediately and the frame submitted will be sent to all connections that are currently actively receiving. When the buffer is no longer needed by the SDK the completion routine is called which tells you that you may now free this buffer. Note that while it is rate, if a connection is stalled and holds onto a buffer until the connection times-out that there is a chance that completions are called out of order.

One can assign a completion handler for asynchronous sending calls, with opaque data that is passed into the completion routine with:

```
void NDIlib_send_set_video_async_completion(NDIlib_send_instance_t p_instance, void*
p_opaque, NDIlib_video_send_async_completion_t p_deallocator);
```

If one passes in a `nullptr` as a completion callback pointer, then the behavior is returned to the default buffer ownership behavior which will lock on a send until the previous call has been asynchronously sent.

It is important to know that the completions will occur on a thread that is called by the NDI SDK, you should ensure that you return quickly from this allocation or free call if you wish to avoid video or audio stalling. Your code should be re-entrant since it might be called from multiple threads at once and should avoid any permanent locks that might occur between sending frames and completion handlers. The handler can be changed out at any time, including at run-time. When you provide a completion handler, the callback is called exactly once for every asynchronous sending call.

When a connection is closed (using `NDIlib_send_destroy`), all outstanding completions will be called before the destroy event is complete.

23.3.1.2 NDI SENDING ON HIGH LATENCY CONNECTIONS

When you are sending NDI data in compressed form (e.g., with `NDIlib_send_send_video_scatter_async` or `NDIlib_send_send_video_async`) to the SDK then the data is transmitted from your buffers without any memory copies. The rules on buffer ownership are that your buffers may be accessed by the SDK until the next asynchronous call. In effect, this means that there will always be a single outstanding NDI compressed send at any time.

On a high latency network, or a network with high packet loss it can take some significant time to be sure that the data being sent has been received by the other side. Because this round-trip-time might be 200ms or more, the NDI SDK needs to have access to your buffer in case any of that data needs to be resent (remember we are trying to work without any memory copies). Since there can only be one outstanding frame being sent by the NDI SDK at once time, when using networks of this kind it might cause poor frame sending performance because we might need to wait for one entire round-trip time to allow the second asynchronous frame to proceed and signal that the buffer is no longer in access by the library.

To best achieve high performance on high latency networks, starting SDK version 5.0.8 the behavior of NDI sending of compressed data has been updated, and the following recommendations apply to sending data.

- When sending compressed frames, the NDI libraries will make copy of the frame so that the caller is not exposed to the round-trip time delays that might slow down the ability to submit frames to the SDK.
- When one specifies an async completion callback using `NDIlib_send_set_video_async_completion` then the SDK will send without any memory copies. When in this mode there will be any number of allowed frames in flight and you will receive a call-back when they are no longer needed.

23.3.2 RECEIVING

NDI receivers are created in the same way as they would be in the NDI SDK, using `NDIlib_recv_create_v3`, or the newer function available to the Advanced SDK, `NDIlib_recv_create_v4`. As with senders, you provide the device name. It is strongly recommended that your device allow its name to be configured so that individual devices can be identified on the network. (This is not currently used by NDI applications, however it is anticipated that it will be in the future and your device will thus be future-proof.)

An example of creating a receiver follows below:

```
// The source to receive from (usually obtained from NDI_find_get_current_sources)
NDIlib_source_t recv_source;
recv_source.p_ndi_name = "Their name"; // The name of the NDI source
recv_source.p_url_address = NULL;

// Setup the structure describing the receiver
NDIlib_recv_create_v3_t recv_create;
recv_create.source_to_connect_to = recv_source;
recv_create.color_format = NDIlib_recv_color_format_compressed; // Compressed pass-through
recv_create.bandwidth = NDIlib_recv_bandwidth_highest; // If you want program quality video
recv_create.allow_video_fields = true;                 // Always true for pass-through
recv_create.p_ndi_name = "Your name";                  // Often configured in your web page

const char *p_config_json = NULL; // You can override the default json file settings
NDIlib_recv_instance_t pRecv = NDIlib_recv_create_v4(&recv_create, p_config_json);
```

It is crucial that you provide an XML identification for all hardware devices. This should include your vendor name, model number, serial number and firmware version.

For example, the following XML identification would work (although it should be filled in with the correct settings for your device (serial numbers should be unique to each device manufactured):

```
NDIlib_metadata_frame_t NDI_product_type;
NDI_product_type.p_data = "<ndi_product long_name=\"NDILib Recv Example.\" "
                          "             short_name=\"NDILib Recv\" "
                          "             manufacturer=\"CoolCo, inc.\" "
                          "             version=\"1.000.000\" "
                          "             session=\"default\" "
                          "             model_name=\"S1\" "
                          "             serial=\"ABCDEFG\" />";
NDIlib_recv_add_connection_metadata(pRecv, &NDI_product_type);
```

Note that you can create a receiver for the preview stream in addition to the program stream. Preview streams might be beneficial for picture-in-picture support, for example.

To do so, you would specify `NDIlib_recv_bandwidth_lowest` in the bandwidth field of the create struct, rather than `NDIlib_recv_bandwidth_highest`. You can capture audio with a program or preview stream receiver, or you can create a dedicated audio-only receiver and capture audio in a loop on another thread.

It is recommended that you also provide a “preferred” video format that you want to an up-stream device, since this has been commonly used by NDI applications. For instance, CG applications can be informed of a resolution you would like, and will automatically configure themselves to this resolution. An example might be as follows:

```
NDIlib_metadata_frame_t NDI_format_type;
NDI_format_type.p_data = "<ndi_format>"
                         "  <video_format xres=\"1920\" yres=\"1080\" "
                         "                frame_rate_n=\"60000\" frame_rate_d=\"1001\"
                         "                aspect_ratio=\"1.77778\" progressive=\"true\"/>"
                         "   <audio_format no_channels=\"4\" sample_rate=\"48000\"/>"
                         "</ndi_format>";

NDIlib_recv_add_connection_metadata(pSend, &NDI_format_type);
```

Here is a simple example of a loop capturing video only from the program stream receiver:

```
while (true)
{
    // Capture a frame of video
    NDIlib_video_frame_v2_t frame;
    if (NDIlib_frame_type_video == NDIlib_recv_capture_v3(pRecv, &frame, NULL, NULL, 100))
    {
        switch (frame.FourCC)
        {
            case (NDIlib_frame_format_type_e)NDIlib_FourCC_type_SHQ0_highest_bandwidth:
                // Decode the frame.p_data SpeedHQ 420 buffer here
                break;
            case (NDIlib_frame_format_type_e)NDIlib_FourCC_type_SHQ2_highest_bandwidth:
                // Decode the frame.p_data SpeedHQ 422 buffer here
                break;
            case (NDIlib_frame_format_type_e)NDIlib_FourCC_type_SHQ7_highest_bandwidth:
                // Decode the frame.p_data SpeedHQ 4224 buffer here
                break;
        }

        NDIlib_recv_free_video_v2(pRecv, &frame);
    }
}
```

Hint: If desired, you could asynchronously capture and decode by moving decoding to another thread. Just don't forget to use `NDIlib_recv_free_video_v2` to free the video buffers when you finish decoding.

The `line_stride_in_bytes` field will be used to tell you the size in bytes of the compressed video data.

23.3.2.1 CUSTOM ALLOCATORS

The Advanced SDK allows you to provide custom memory allocators for receiving audio and video frames either in compressed or uncompressed formats. This allows you to ask NDI to decompress or receive data into a buffer allocated by you own application.

Some possible use cases for this are:

- Decompress into your own memory buffers for use in inter-process memory sharing; when you might want one or more NDI inputs to exist in their own process.
- You wish to fill in buffers that might be more optimally accessible by dedicated hardware, including GPUs.
- Minimizing memory copies does to the internal structure of your own application.
- Allocating a buffer with memory alignment that matches your need.
- This allows you to enter your own video or audio stride, allowing you to have NDI provide buffers that closely match the needs of your own application.

Note: It is often assumed that decompressing into a GPU accessible buffer will yield improved performance, however often these buffers are allocated using write combining memory that is not commonly CPU cacheable. Often decoding into these buffers is slower than decoding them into a regular memory buffer and performing a memcpy that is specifically optimized for copying into write combining memory,

If you are going to simply use your own pooled memory allocator, it is unlikely that it will give you significant performance enhancements over what the NDI SDK already has implemented internally. The NDI SDK uses a lock-free memory pooling mechanism to offer very quick frame allocation.

It is important to know that the allocations will occur on a thread that is called by the NDI SDK, you should ensure that you return quickly from this allocation or free call if you wish to avoid video or audio stalling. Your code should be re-entrant since it might be called from multiple threads at once. The allocators can be changed out at any time, including at run-time. When you provide an allocation and free callback, the free callback is called exactly once for every allocation call, even if the allocators are changed out. There are some error conditions under which it is possible that a frame will be allocated, but not returned by the SDK since there might be a network or data integrity problem. In this case the frame is simply freed using your customer de-allocator.

A. VIDEO ALLOCATORS

In order to replace a memory allocator, one should implement two functions with one to allocate a new video frame and set the stride, and one to free that frame when the SDK no longer needs it. If you are implementing a memory allocator for use with uncompressed video frames, you should check the requested frame format within the allocator and fill in the `p_buffer` and `line_stride_in_bytes` members with the `NDIlib_video_frame_v2_t` structure that is passed into the function. When the SDK is free with the frame the corresponding de-allocation function will be called and you should free the `p_buffer` member using whatever means you might need.

The `p_opaque` pointer that may be passed into the function setup is your own custom data that will then be passed each time that an allocator or de-allocator is called.

An example uncompressed video frame allocator might look as follows. Note that you need not always support allocating under all possible frame formats since your allocator will only be called for the formats specified when receiving video with the SDK. This example is provided to illustrate how all formats might be used.

```
bool video_custom_allocator(void* p_opaque, NDIlib_video_frame_v2_t* p_video_data)
{       switch (p_video_data->FourCC)
        {       case NDIlib_FourCC_video_type_UYVY:
                        p_video_data->line_stride_in_bytes=p_video_data->xres*2;
                        p_video_data->p_data=(uint8_t*)::malloc(p_video_data
                                ->line_stride_in_bytes*p_video_data->yres);
                        break;

                case NDIlib_FourCC_video_type_UYVA:
                        p_video_data->line_stride_in_bytes=p_video_data->xres*2;
                        p_video_data->p_data=(uint8_t*)::malloc(p_video_data
                                        ->line_stride_in_bytes*p_video_data->yres +
                            /* Alpha */p_video_data->line_stride_in_bytes / 2*p_video_data->yres);
                        break;

                case NDIlib_FourCC_video_type_P216:
                        p_video_data->line_stride_in_bytes=p_video_data->xres*2*sizeof(int16_t);
                        p_video_data->p_data=(uint8_t*)::malloc(p_video_data
                                        ->line_stride_in_bytes*p_video_data->yres);
                        break;

                case NDIlib_FourCC_video_type_PA16:
                        p_video_data->line_stride_in_bytes=p_video_data->xres*2*sizeof(int16_t);
                        p_video_data->p_data=(uint8_t*)::malloc(p_video_data
                                        ->line_stride_in_bytes*p_video_data->yres +
                           /* Alpha */p_video_data->line_stride_in_bytes / 2*p_video_data->yres);
                        break;

                case NDIlib_FourCC_video_type_BGRA:
                case NDIlib_FourCC_video_type_BGRX:
                case NDIlib_FourCC_video_type_RGBA:
                case NDIlib_FourCC_video_type_RGBX:
                        p_video_data->line_stride_in_bytes=p_video_data->xres*4;
                        p_video_data->p_data=(uint8_t*)::malloc(p_video_data
                                        ->line_stride_in_bytes*p_video_data->yres);
                        break;

                default:
                        // Error, not a supported FourCC
                        p_video_data->line_stride_in_bytes=0;
                        p_video_data->p_data=nullptr;
                        return false;
        }

        // Success
        return true;
}

bool video_custom_deallocator(void* p_opaque, const NDIlib_video_frame_v2_t* p_video_data)
{       ::free(p_video_data->p_data);
```

```
        // Success
        return true;
}
```

One may then simply assign the allocator for any receiver with:

```
NDIlib_recv_set_video_allocator(pNDI_recv, nullptr, video_custom_allocator,
                                                    video_custom_deallocator);
```

If you wish to reset the video memory allocators, at any time you may simply pass in null pointers:

```
NDIlib_recv_set_video_allocator(pNDI_recv, nullptr, nullptr, nullptr);
```

As a final note, although it is not needed that often, it is possible to use your own memory allocators to receive compressed video format if the NDI receiver is being specified to receive compressed data

B. AUDIO ALLOCATORS

Audio allocations are implemented almost identically to video allocations, simply with different FourCC codes. An example memory allocator for audio might look as follows:

```
bool audio_custom_allocator(void* p_opaque, NDIlib_audio_frame_v3_t* p_audio_data)
{       // Allocate uncompressed audio
        switch (p_audio_data->FourCC)
        {       case NDIlib_FourCC_audio_type_FLTP:
                        p_audio_data->channel_stride_in_bytes=sizeof(float)*p_audio_data
                                ->no_samples;
                        p_audio_data->p_data=(uint8_t*)::malloc(p_audio_data
                                ->channel_stride_in_bytes * p_audio_data->no_channels);
                        break;

                default:
                        p_audio_data->channel_stride_in_bytes = 0;
                        p_audio_data->p_data = nullptr;
                        return false;
        };

        // Success
        return true;
}

bool audio_custom_deallocator(void* p_opaque, const NDIlib_audio_frame_v3_t* p_audio_data)
{       // Free the memory
        ::free(p_audio_data->p_data);

        // Success
        return true;
}
```

One may then simply assign the allocator for any receiver with:

```
NDIlib_recv_set_audio_allocator(pNDI_recv, nullptr, audio_custom_allocator,
                                                    audio_custom_deallocator);
```

If you wish to reset the video memory allocators, at any time you may simply pass in null pointers:

```
NDIlib_recv_set_audio_allocator(pNDI_recv, nullptr, nullptr, nullptr);
```

23.3.3 FINDING

As documented elsewhere, the NDI filter allows you to locate all NDI sources on the network. When using a discovery server, the Advanced SDK allows you to specify per connection metadata by specifying a "source.metadata" field in the sender JSON (see documentation for configuration files). When using an NDI finder one can receive the list of all sources on the network with their associated metadata using the function:

```
const NDIlib_source_v2_t* NDIlib_find_get_current_sources_v2(NDIlib_find_instance_t
p_instance, uint32_t* p_no_sources);
```

This function returns the list of sources, including their metadata in a fashion that is identical to the regular `NDIlib_find_get_current_sources` function.

23.3.4 VIDEO FORMATS

Decoding is more complex than encoding, because you do not get to specify what format you are sent. It is recommended that you support as many possible video formats as you wish, although if this is not possible (e.g. non video resolutions) you can either scale, or simply provide a place-holder image. It is important that you support the three possible video formats (4:2:0, 4:2:2, 4:2:2:4) since these are in common use. If you cannot process the alpha channel it is recommended that you multiply the image against black.

It is also important to understand that it is the NDI sender that determines the video and audio clock rates. A simple frame-buffer is not sufficient to smoothly display audio and video without glitches.

23.3.4.1 RECEIVER CODEC SUPPORT LEVEL

When creating an NDI receiver to receive compressed data, it is very important to specify the `color_format` field correctly on the `NDIlib_recv_create_v3_t` structure. The following table will list all of the available values introduced with the Advanced SDK and what the values mean. If you specify a value but do not know how to handle certain frame types, it is very important that you check the `FourCC` of the frame and discard accordingly.

Audio for all formats, except the ones with the `with_audio` suffix, will be delivered in floating-point format. If the NDI source is sending AAC audio, the NDI library will attempt to decompress the audio frame and also return that as floating-point format.

color_format value	Description
NDIlib_recv_color_format_compressed	This value has the same meaning as NDIlib_recv_color_format_compressed_v1.
NDIlib_recv_color_format_compressed_v1	When connected to an NDI source that is sending SpeedHQ video, the compressed frames will be delivered to you. This mode assumes you only know how to handle with SpeedHQ frames and no other format, not even uncompressed. If the NDI source sends any other video compression format frames will not be delivered to you, nor will there be an attempt to decompress the frames in the NDI library.

NDIlib_recv_color_format_compressed_v2	When connected to an NDI source that is sending SpeedHQ video, the compressed frames will be delivered to you. This mode assumes you only know how to handle SpeedHQ frames and uncompressed frames, but no other format. If the NDI source is sending any other compression format, the frames will be delivered in UYVY format if the NDI library could decompress it.
NDIlib_recv_color_format_compressed_v3	When connected to an NDI source that is sending SpeedHQ video or H.264 video, the compressed frames will be delivered to you. This mode assumes you know how to handle SpeedHQ, H.264, and uncompressed frames, but no other format. If the NDI source is sending any other video compression format, they will be delivered to you in UYVY format if the NDI library can decompress it.
NDIlib_recv_color_format_compressed_v3_with_audio	This value has the same meaning as NDIlib_recv_color_format_compressed_v3, but allows AAC audio frames to be passed to your layer without the NDI library decompressing them. If the NDI source is not sending AAC audio, then you will receive audio in floating-point format.
NDIlib_recv_color_format_compressed_v4	When connected to an NDI source that is sending SpeedHQ, H.264 or H.265 video, the compressed frames will be delivered to you. This mode assumes you know how to handle SpeedHQ, H.264, H.265, and uncompressed frames, but no other format. If the NDI source is sending any other video compression format, they will be delivered to you in UYVY format if the NDI library can decompress it.
NDIlib_recv_color_format_compressed_v4_with_audio	This value has the same meaning as NDIlib_recv_color_format_compressed_v4, but allows AAC audio frames to be passed to your layer without the NDI library decompressing them. If the NDI source is not sending AAC audio, then you will receive audio in floating-point format.

23.3.4.2 FRAME SYNCHRONIZATION

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

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

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

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

A "time base corrector" (TBC) or frame-synchronizer for the video clock provides another mechanism to handle these issues. This approach uses hysteresis to determine the best time to either drop or insert a video frame to achieve smooth video playback (audio should be dynamically sampled with a high order resampling filter to adaptively track clocking differences).

It's quite difficult to develop something that is correct for all scenarios, so the NDI SDK provides an implementation to help you develop real time audio/video applications without assuming responsibility for the significant complexity involved. Another way to view what this component of the SDK does is to think of it as transforming 'push' sources (i.e. NDI sources in which the data is pushed from the sender to the receiver) into 'pull' sources, wherein the host application pulls the data down-stream. The frame-sync automatically tracks all clocks to achieve the best video and audio performance while doing so.

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

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

Scenario	Recommendation
Video playback on screen or a multiviewer	Yes – you want the clock to be synced with vertical refresh. On a multi-viewer you would have a frame-sync for every video source, then call all of them on each v-sync and redraw all sources at that time.
Audio playback through sound card	Yes – the clock should be synced with your sound card clock.
Video mixing of sources	Yes – all video input clocks need to be synced to your output video clock. You can take each of the video inputs and frame-synchronize them together.
Audio mixing	Yes – you want all input audio clocks to be brought into sync with your output audio clock. You would create a frame-synchronizer for each audio source and – when driving the output – call each one, asking for the correct number of samples and sample-rate for your output.

Recording a single channel	No – you should record the signal in the raw form without any re-clocking.
Recording multiple channels	Maybe – If you want to sync some input channels to match a master clock so 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 on 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 so that you have the most correct possible field ordering on output.

- The same frame can be returned multiple times if duplication is needed to match the timing criteria.

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

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

```
void NDIlib_framesync_capture_video(
    NDIlib_framesync_instance_t p_instance, // The frame-sync instance
    NDIlib_video_frame_v2_t* p_video_data,  // The destination video frame
    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);
```

23.4 GENLOCK

When using NDI to send video onto the network it is very common that one uses the computer clock to know what speed to send frames at; indeed when specifying `clocked=true` within the NDI sender the SDK will use the system clock in order to pace the sending of frames for you.

It is very common in video systems that you wish to make sure that all your NDI sources are synchronized together so that they all send video at the exact same rate and the same times. While it is tempting to solve this by having a very high precision “reference clock” (e.g. PTP) this often works very well on local networks but does not easily extend to systems that remote from each-other (e.g. your local network and in the WAN).

The NDI SDK now offers a way to easily allow you to clock any number of video sources on the network to match a centralized clock, and even interface those with external video clocks like local SDI sources which are commonly used as a genlock signal.

The NDI genlock allows one to create a “genlock clock” which is attached to any NDI sender on the network. That genlock clock can then be used to correctly time all senders on the network so that they are correctly timed with the NDI sender. By sharing this NDI source into the cloud, you can ensure that you have full genlock support that spans both on-premise, remote networks and in-cloud connections. By driving an NDI source using an SDI (or PTP, 2110, HDMI) converter it is even simple to genlock your entire NDI network to a physical genlock signal.

For an NDI source to correctly be able to operate as an NDI Genlock, it is important to bear in mind a couple of key ingredients as outlined below.

- It is very strongly recommended that the NDI source is a stream from NDI version 5 which has been significantly improved to support genlock capabilities. It is possible that some NDI streams from previous versions are not fully compliant with how NDI genlock operates.
- It is important that the source has enough network bandwidth to drive a reduced bandwidth signal to all the NDI genlock instances. Some Advanced SDK NDI converters might fail in this regard in which case an NDI Proxy may be used to relay the signal (and might also be used to relay cloud genlock as well). Configuring this source for multicast might also help, although multicast often is hard to full support.
- NDI genlock is very robust and supports correct cross-frame-rate locking. For instance, a sender might be 30Hz and you are genlocking a 60Hz signal to it. This is however not a recommended workflow where it can be avoided.

- Some NDI sources like Test Pattern generator and NDI Screen Capture do not always send a regular stream of frames. They do this in order to save network bandwidth and CPU time. Sources such as these cannot be used as a basis for genlock.
- Remember that when creating NDI senders that you wish to use with genlock that you specify the `use_clock` values as false when the senders are created, if not then the system clock will still be applied at the NDI sender level.
- If the genlock clock cannot correctly genlock to an NDI sender for some reason it will fall back to using the system clock and so can continue to work reasonably.
- Since there is some (low) overhead associated with each genlock instance it is recommended that you only have one for each source that you wish to lock too.

To create an NDI genlock instance, you use the function `NDIlib_genlock_create`, you may specify the NDI source that you wish to lock the signal too, and the NDI JSON settings associated with it. It is important that you remember to fill out the `vendor_id` in the JSON or this sender will have the same limitation as NDI receivers. It is legal to create an NDI genlock that has a null source name and then later use the `NDIlib_genlock_connect` function to change the connection being used. Like all other NDI SDK functions, a genlock object can be disposed with the function `NDIlib_genlock_destroy`.

If you wish to change the NDI source that is being used for the genlock, one can call the function `NDIlib_genlock_connect`. If the parameter is `nullptr` then the NDI source will be disconnected and the genlock operation will fall back to using the system clock.

In order to determine if a particular NDI source is correctly operating as a genlock signal one may call `NDIlib_genlock_is_active`. If the NDI sender that is being used as the genlock source is not currently sending an active signal then this will return `false`. Note that the functions to perform the clocking operation return whether the genlock is active and so this function need not be polled within a sending loop and is provided as a convenience.

Once you have created an NDI genlock instance and have it locked to an NDI receiver, then one may simply call the functions `NDIlib_genlock_wait_video` and `NDIlib_genlock_wait_audio`. These functions will wait until it is time for the next video or audio frame to be delivered and then return so that you can send it. While these structures take in a full frame descriptor, only the minimum number of members are used and so the rest need not be filled in. For video a `NDIlib_video_frame_v2_t` is used, however only the `frame_rate_N, frame_rate_D` and `frame_format_type` members need be valid. For audio a `NDIlib_audio_frame_v3_t` is used, however only the `no_samples` and `sample_rate` need be valid. These functions return a `bool` value which will tell you whether the source is currently genlocked or whether there is no signal and so the system clock has been used for timing.

The audio and video waiting functions are entirely thread safe and you may have a separate thread for the timing and of video and audio as needed.

An example application is given below that shows how one might lock to an NDI source and then send frames that would match the clock of that NDI source.

```
// We are going to start by creating an NDI genlock instance
NDIlib_source_t src_name(< Insert your NDI source here >);
NDIlib_genlock_instance_t p_genlock = NDIlib_genlock_create(&src_name,
                        nullptr/* Note that your vendor JSON is required here */);

// We are now going to loop and genlock to the signal at 59.94Hz
while (<Your application is running>)
```

```
{   // Setup the frame header.
    NDIlib_video_frame_v2_t frame;
    frame.frame_rate_N = 30000;
    frame.frame_rate_D = 1001;
    frame.frame_format_type = NDIlib_frame_format_type_progressive;

    // We wait for a frame to send
    const bool genlock_locked = NDIlib_genlock_wait_video(p_genlock, &frame);
    printf("Send a frame, currently locked to %s.\n", genlock_locked ? "remote source"
                                                                     : "system clock");
}

NDIlib_genlock_destroy(p_genlock);
```

23.5 AV SYNC

NDI relies on time-stamps to synchronize incoming audio and video. You can of course fill these in yourself although by default NDI will use the NTP time so that even if multiple streams are coming from different machines that you make then synchronize them together.

This API allows you to synchronize audio and video from either the same or different NDI sources with accuracy approaching one audio sample as long as a sender uses sample-accurate time-stamps. While this of course can be achieved at a regular API user level this is a non-trivial task (for instance by ensuring that you audio and video frames have the same exact time-stamps), with this API helping solve at least the following problems in a relatively simple way:

- The audio and video streams might be generated by applications that are not generating nanosecond accurate time-stamps and so a significant level of filtering and accuracy improvement is needed so that frames can be aligned with sample-level accuracy.
- Packetizing the audio so that it correctly matches the video frames exactly is complex, and preserving and computing the correct timestamps, metadata and timecodes in this process is non-trivial.
- Often a particular pattern of output samples is needed for interfacing with hardware is needed (e.g. NTSC often requires audio samples on frames in alternating patterns of 1601 and 1602 samples) and it is very beneficial to ask the API for the number of samples wanted and it correctly aligns audio to this patter.
- Correctly detecting when a stream has no audio or there are format changes requires special consideration.
- Timeouts and ensuring that errors remain within the Nyquist sampling limits when running on a computer that might already be under load or have inaccurate clocks makes a solution to this challenging.

This API is a little more complex than some of the others in this SDK because it allows for great flexibility within just a few functions. In particular please play close attention to the exact definition of the return results and how they work.

23.5.1 GUIDELINES

This API will take audio for a `NDIlib_recv_instance_t` instance, returning a `NDIlib_avsync_instance_t` object. It will then look at the audio being received on this input. One may then take video frames from either the same `NDIlib_recv_instance_t` instance or a different one and one may query the `NDIlib_avsync_instance_t` for the audio that matches that video frame. This takes the time-stamps that are attached to each frame to synchronize the audio and video with high prevision.

It is important when working with this object that you ensure that the following conditions are met for the best results.

- The audio and video do not need to be on exactly the same clock (i.e. they do not need to be "genlocked"). With this said, if they are not then the number of audio samples that are returned with the video frame might not be exactly what is expected and might vary over time. For instance, if the audio being received is running on a clock that is 1% faster than the video clock then you will receive the exactly synchronized audio although there will be 1% more audio samples than one might expect.
- The audio and video are expected to behave *relatively* well (but not perfectly). If there are significant gaps in the audio or video streams, or the streams are delivered such that they have very significant jitter (well beyond the Nyquist sampling limit) then it is hard to ensure that they are reconstructed perfectly although all effort is made to act gracefully in these situations. As an example, if audio is delivered significantly later than the corresponding video then this device will end up needing to run behind on the video to find the exactly matching audio.
- You may pass an `NDIlib_recv_instance_t` into `NDIlib_avsync_create` and the returned `NDIlib_avsync_instance_t` will capture the audio from the device. You may simultaneously use this same `NDIlib_recv_instance_t` to capture the audio and use the corresponding `NDIlib_avsync_instance_t` as the means of synchronizing the audio and video from the same device. You similarly capture audio and video from different devices if you wish to use an `NDIlib_avsync_instance_t` to synchronize audio and video from different sources, however note the comments above on the clocking from disparate sources.
- This implementation will only support and process audio that is supported on your platform by the decoders within NDI (e.g. on Linux AAC audio might not correctly be used with this implementation).

23.5.2 CREATING AND DESTROYING DEVICES

It is very simple to create and destroy a device. Simply create an NDI receiver that one wishes to receive audio from and pass it into `NDIlib_avsync_create`, this will return an instance of type `NDIlib_avsync_instance_t` which may be used until you no longer need it. Once you no longer need it you should call `NDIlib_avsync_destroy`. Please note that you should destroy a device before the corresponding `NDIlib_recv_instance_t` is called since an internal reference to this object is kept within the `NDIlib_avsync_instance_t`.

23.5.3 RECOVERING AUDIO

The most typical use of the synchronizing function would be to pass capture a video frame by the means that you normally would, then to make a call to:

```
NDIlib_avsync_ret_e NDIlib_avsync_syncronize(NDIlib_avsync_instance_t p_avsync,
                                             const NDIlib_video_frame_v2_t *p_video_frame,
                                             NDIlib_audio_frame_v3_t* p_audio_frame);
```

This function is however deceptively powerful and the key to understanding this is correctly passing the correct values into the `NDIlib_audio_frame_v3_t* p_audio_frame` parameter of this function. The following describes the values that may be specified.

Parameter	Description
`no_samples`	If this value is "0" when you pass it into the function then if audio may be recovered then the synchronization function will automatically return the exact length of audio that matches the video frame that was passed in as the first parameter. The number of samples returned in this way will almost exactly match the values related to the time-stamps but can be assumed to very accurately reflect the audio that matches the frame.

	Because the timestamps are often subject to noise when frames ate stamped, the number of samples might vary slightly. This will result in a return code of `NDIlib_avsync_ret_success`. If this value is some constant (e.g. 1601 or 1602) then the AV sync will attempt to return this number of samples as long as it is close to the true number of audio samples that are aligned with this frame. This is designed so that you let this class correctly recover audio that might follow some external constraint on the number of samples that are used with video frames. If the number of samples requested is sufficiently close to the number of audio samples that match this frame then a return code of `NDIlib_avsync_ret_success` is returned. If it was not possible to correctly return this number of samples because it did not closely match the number of samples that are truly associated with this video frame then the function will instead return the correct audio (which might be to many or tpo few samples) and return a value `NDIlib_avsync_ret_success_num_samples_not_matched`. It would then be the responsibility of the caller to determine how to best handle this condition. This condition is normally caused by trying to synchronize audio and video those are not on the same clock. When requesting a specific number of audio samples, this is normally computed externally to this function under the assumption of some known audio sample rate. Because incoming audio might change sample rates which would render the number of requested samples invalid please review the section below on how specifying the `sample_rate` parameter of `p_audio_frame`.
`sample_rate`	When this function is “0”, then there is no assumed sample rate and the function will return an audio frame that specifies the current audio sample rate. If you are specifying the number of samples to be captured as non-zero, it is likely that this was computed at a given audio sample-rate. If you specify this on the `p_audio_frame` *as input* then if the sample rate of this audio source does not match it will not capture any audio and will return a result of `NDIlib_avsync_ret_format_changed` and fill in the audio format only in the returned structure. One can then simply recompute the number of required audio samples and simply call the function again to capture the audio with that video frame.

If the `NDIlib_video_frame_v2_t *p_video_frame` is not specified (is `nullptr`) then the audio parameter can be used either to capture all current audio (`no_samples=0`) or a specified number of audio samples (`no_samples` is not zero) and behaves exactly as specified above although all audio is handled and not simply the audio associated with the current video frame.

Please note that this function will fill all parameters of the return frame, including the timecode, timestamp and metadata. The metadata is chosen from the closest matching audio frame and is returned just one time. If the audio frames are all much smaller in duration than the corresponding video frames then some metadata fields might be missing since they no longer have corresponding audio data to be assigned too.

It is important that you call `NDIlib_avsync_free_audio` in order to return the frames returned by `NDIlib_avsync_syncronize`.

The full set of return codes from this function are documented below. Please note that these have integer values which are positive for success and negative for failure.

Error code	Meaning
`*_success`	This function succeeded and returned audio that matches the frame, and if you specified `sample_rate` or `no_samples` then correctly matches those constraints.
`*t_success_num_samples_not_matched`	This function succeeded, but you specified a `no_samples` that could not be matched exactly because this would push the audio and video frame sufficiently out of alignment. The full audio samples associated with this video frame are returned and it does not match `no_samples`. It might be a higher or lower number.
`*_no_audio_stream_received`	This indicates that there is currently no audio stream and so it would not be possible to return any audio data. This might indicate a video only stream.
`*_ret_no_samples_found`	This indicates that this video frame did not have matching audio data. This might be because the time at which this video frame was sent there was no corresponding audio. It might also indicate that the sending of the audio and video streams is sufficiently unaligned that they cannot easily be resynchronized; for instance the audio data arriving is more than a second out of sync with the corresponding video data.
`*_format_changed`	A `sample_rate and no_samples` was specified, however the sample rate did not match and so this function has returned and correctly filled in the
`*_ret_internal_error`	This function was called with incorrect

23.6 USING H.264, H.265, AND AAC CODECS

We recommend that you start by reviewing the sending of frames over the network using the NDI® SDK. The Advanced SDK is designed to operate almost identically, although you are able to able to send compressed data streams directly.

Currently the Advanced SDK supports H.264 or H.265 compression at a very wide variety of bitrates, resolutions, and framerates; and it would be common for you to use hardware-assisted compression to generate the compressed video stream on Advanced SDK devices, then send this onto the network using the NDI Advanced SDK. AAC audio is supported for audio transmission.

To send a compressed video frame, you should use the Advanced SDK structure `NDIlib_compressed_packet_t` to pack your data. You should allocate memory for your packet so that the following data will be in a single block:

Size and Type	Name	Details
uint32_t, 4 bytes	version	This represents the current version number of the structure. This should be set to `NDIlib_compressed_packet_t:: version_0`, which has a value of 44 currently (representing the structure size).
uint32_t, 4 bytes	fourCC	This is the FourCC for the current compression format. Currently H.264 is supported, although other formats might be available in the future. • H.264 should be specified using `NDIlib_FourCC_type_H264`. • H.265 should be specified using `NDIlib_FourCC_type_HEVC`. • AAC audio should be submitted using `NDIlib_FourCC_type_AAC`.
int64_t, 8 bytes	pts	The stream presentation time stamp. See notes in the next section.
int64_t, 8 bytes	dts	The stream display time stamp. See notes in the next section.
int64_t, 8 bytes	reserved	This is currently a reserved field, and will not be propagated by the SDK.
uint32_t, 4 bytes	flags	The flags that apply to this frame. Currently there are two supported values for this setting: • `NDIlib_compressed_packet_t::flags_none`. Nothing. • `NDIlib_compressed_packet_t::flags_keyframe`. This is a frame that can be decoded without dependence on other stream data. This normally means that this is considered an I-Frame. For H.264 and H.265, key-frames must have extra-data. This should always be set for AAC audio.
uint32_t, 4 bytes	data_size	The size of the compressed video frame.
uint32_t, 4 bytes	extra_data_size	The size of the ancillary extra data for the current codec settings. This is required for key-frames in most compressed video formats.
data_size bytes	[data]	This is the compressed video frame data in byte format.
extra_data_size bytes	[extra_data]	This is the compressed ancillary data if extra_data_size is not zero.

23.6.1 SENDING AUDIO FRAMES

To submit audio frames, start by building a structure of type `NDIlib_compressed_packet_t` that has the compressed AAC audio data within it.

The following example creates a compressed audio frame when you have compressed audio data of size `audio_data_size`, at pointer `p_audio_data` with `audio_extra_data_size`[3] at pointer `p_audio_extra_data`:

[3] As noted in the AAC support section of this document, this would almost always be two bytes.

```
// See notes above
uint8_t* p_audio_data;
uint32_t audio_data_size;

// See notes above
uint8_t* p_audio_extra_data;
uint32_t audio_extra_data_size;

// Compute the total size of the structure
uint32_t packet_size = sizeof(NDIlib_compressed_packet_t) + audio_data_size +
                                                            audio_extra_data_size;

// Allocate the structure
NDIlib_compressed_packet_t* p_packet = (NDIlib_compressed_packet_t*)malloc(packet_size);

// Fill in the settings
p_packet->version = NDIlib_compressed_packet_t::version_0;
p_packet->fourCC = NDIlib_FourCC_type_AAC;
p_packet->pts = 0; // These should be filled in correctly if possible.
p_packet->dts = 0;
p_packet->flags = NDIlib_compressed_packet_t::flags_keyframe; // All AAC packets are a
keyframe
p_packet->data_size = audio_data_size;
p_packet->extra_data_size = audio_extra_data_size;

// Compute the pointer to the compressed audio data, then copy the memory into place.
uint8_t* p_dst_audio_data = (uint8_t*)(1 + p_packet);
memcpy(p_dst_audio_data, p_audio_data, audio_data_size);

// Compute the pointer to the ancillary extra data
uint8_t* p_dst_extra_audio_data = p_dst_audio_data + audio_data_size;
memcpy(p_dst_extra_audio_data, p_audio_extra_data, audio_extra_data_size);
```

Once you have the compressed data structure that describes the frames, then you need simply create a regular `NDIlib_audio_frame_v3_t` to pass to NDI SDK functions as shown in the following example:

```
// Create a regular NDI audio frame, but of compressed format
NDIlib_audio_frame_v3_t audio_frame;
audio_frame.sample_rate = 48000;               // This must match AAC format
audio_frame.no_channels = 2;                   // This must match AAC format
audio_frame.no_samples = 1024;                 // This must match AAC format
audio_frame.timecode = p_packet->pts;          // Might be a good value. Read docs!
audio_frame.FourCC = (NDIlib_FourCC_audio_type_e)NDIlib_FourCC_audio_type_ex_AAC;
audio_frame.p_data = (uint8_t*)p_packet;
audio_frame.data_size_in_bytes = packet_size;
audio_frame.p_metadata = "<something/>";       // Per frame metadata is fully supported.

// Transmit the AAC audio data to NDI
NDIlib_send_send_audio_v3(pSender, &audio_frame);
```

Once this is sent the audio data will be transmitted, and you may free or re-use any audio data pointers that you allocated to represent the data. It is obviously possible to use a pool of memory to build audio packets without per-packet memory allocations.

The send audio function is thread-safe, and may be called on a separate thread from compression transmission.

23.6.2 SENDING VIDEO FRAMES

Because you are undertaking all compression of the video yourself, you should provide a full bandwidth and a preview bandwidth video stream. The full bandwidth stream may be any resolution and framerate; the preview bandwidth should be 640 pixels wide and square pixels (e.g. 640x360 pixels when at 16:9 image aspect ratio).

You will then need to submit these two frames separately to the SDK. The creation of the frames is identical to the audio example in the previous section. The following is an example of just one of the two required streams:

```
uint8_t* p_h264_data;
uint32_t h264_data_size;

// This is probably zero for non I-frames, but MUST be set of I-frames
uint8_t* p_h264_extra_data;
uint32_t h264_extra_data_size;

// Compute the total size of the structure
uint32_t packet_size = sizeof(NDIlib_compressed_packet_t) + h264_data_size +
                                                            h264_extra_data_size;

// Allocate the structure
NDIlib_compressed_packet_t* p_packet = (NDIlib_compressed_packet_t*)malloc(packet_size);

// Fill in the settings
p_packet->version = NDIlib_compressed_packet_t::version_0;
p_packet->fourCC = NDIlib_FourCC_type_H264;
p_packet->pts = 0;       // These should be filled in correctly !
p_packet->dts = 0;
p_packet->flags = is_I_frame ? NDIlib_compressed_packet_t::flags_keyframe
                       : NDIlib_compressed_packet_t::flags_none;
p_packet->data_size = h264_data_size;
p_packet->extra_data_size = h264_extra_data_size;

// Compute the pointer to the compressed h264 data, then copy the memory into place.
uint8_t* p_dst_h264_data = (uint8_t*)(1 + p_packet);
memcpy(p_dst_h264_data, p_h264_data, h264_data_size);

// Compute the pointer to the ancillary extra data
uint8_t* p_dst_extra_h264_data = p_dst_h264_data + h264_data_size;
memcpy(p_dst_extra_h264_data, p_h264_extra_data, h264_extra_data_size);
```

Having filled in the audio compressed data structures, fill in a video header as shown in the following example.

Note: It is crucial that the value of the FourCC specifies whether this is the full or preview resolution streams.

```
// Create a regular NDI audio frame, but of compressed format
NDIlib_video_frame_v2_t video_frame;
video_frame.xres = 1920; // Must match H.264 data
video_frame.yres = 1080; // Must match H.264 data

// This must value is dependent on whether this is a full or preview
// stream. This example shows the full resolution stream, the preview
// resolution stream would specify NDIlib_FourCC_type_H264_lowest_bandwidth
video_frame.FourCC = (NDIlib_FourCC_video_type_e)NDIlib_FourCC_type_H264_highest_bandwidth;

// Any reasonable sample rate is supported
video_frame.frame_rate_N = 60000;
video_frame.frame_rate_D = 1001;

// Any reasonable aspect ratio is supported
```

```
    video_frame.picture_aspect_ratio = 16.0f/9.0f;

    // We only currently allow
    video_frame.frame_format_type = NDIlib_frame_format_type_progressive;

    // Choose a good value, this is an example
    video_frame.timecode = p_packet->pts;

    // Set the data
    video_frame.p_data = (uint8_t*)p_packet;
    video_frame.data_size_in_bytes = packet_size;

    // Metadata is supported
    video_frame.p_metadata = "<Hello/>";

    // Transmit the H.264 video data to NDI, async is fully supported.
    // But read SDK description of buffer life-times.
    NDIlib_send_send_video_async_v2(pSender, &video_frame);
```

A critical part of NDI is that you ask the SDK if you should insert an I-frame. This can be achieved by making a call to `NDIlib_send_is_keyframe_required`. This call returns true when an I-Frame should be inserted in order for a down-stream source to correctly decode the image without errors.

For instance, when there is a new NDI connection this function will return true; when a down-stream source has dropped a packet and can no longer decode the rest of the GOP, then this will be detected and return true, etc.

You are free to insert I-frames when you want based on your GOP requirements, but when this function returns true then you should issue an I-Frame at the next possible time. This is required functionality for a good user experience and a compliant NDI HX source. The stream validation (described later) will verify that this practice is followed.

Although you can determine your own bitrates for H.264 or H.265 streams, the NDI HX SDK will also provide guidance if you wish to call `NDIlib_send_get_target_frame_size`. When used with H.264 or H.265 FourCC's in the structures, this will provide a reasonable *average* bitrate recommendation based on the selected resolution, framerate, etc. This provides estimates for all combinations of framerates and resolutions, but compliance is not required.

Please note that NDI is a real-time API, meaning that you should make all effort to pass off video and audio as they arrive in synchronization with each-other. These are passed through the transmission layers downstream to the remote device with the lowest possible latency, and may be received by sources that are observing one or both streams (when possible, bandwidth is only allocated for the used streams in transmission).

Audio and video are sent when they are passed to the API, allowing one or both streams to stop or start as needed at any layer. As a result, frames are not "held" in order to synchronize streams, resulting in higher performance.

23.6.3 H.264 SUPPORT

23.6.3.1 SUPPORTED FORMATS

NDI assumes that all H.264 data is as specified in Annex B of ITU-T H.264 | ISO/IEC 14496-10. The data must include the start codes. We support 4:2:0 in Baseline, Main, and High profiles up to level 5.1. We recommend that a I-Frame interval of between one and two seconds; it is however very important that you recall that the SDK will inform you when I-Frame insertions are required. The recommended bit-rate of the stream is provided by `NDIlib_send_get_target_bit_rate`; for HX it is considered reasonable that you allow the user to select whether you wish a bit-rate in the range of 0.1 – 2.0x the bit-rate provided by this function.

The current NDI implementation supports H.264 decoding without any installed plugins across the Windows and macOS targets. If you require Linux support, please contact NDI SDK support.

Full hardware acceleration is provided on these platforms if the relevant hardware acceleration recommendations are provided by an end user application as described in the NDI Advanced SDK documentation. On Windows 7, the maximum decoder resolution is 1920x1080 due to OS limitations. On other platforms the maximum resolution is currently 4096x2304.

While you may use all H.264 frame types, we recommended considering the use of B frames with caution, since these will cause increased decoder latency in some configurations. Because NDI is a low latency, real-time scheme, it is crucial that you focus great effort on achieving low latency and high quality.

It is crucial that your codec provide reasonable accurately-timed frames; a common problem that we have observed is that – with some bitrate control algorithms – that the I-Frames are sufficiently large that subsequent P frames are delayed far in excess of the Nyquist sampling limit, which tends to cause jittery video transmission.

23.6.3.2 EXTRA DATA

The extra ancillary data required to configure an H.264 codec must correctly be provided. This must exist for all I-frames. The structure of this data should contain concatenated NAL units in Annex B format, along with their start codes. For H.264, they are the SPS and PPS NAL units.

23.6.3.3 PTS AND DTS

The PTS and DTS are not used directly by the NDI SDK, however they are important in order to ensure that frames may be decoded and displayed in correct order. While we recommend that you use 100 ns intervals for these values, any time-base is technically supported as long as the ordering of integers is correct.

23.6.4 H.265 SUPPORT

23.6.4.1 SUPPORTED FORMATS

NDI assumes that all H.265 data is as specified in Annex B of ITU-T H.265 | ISO/IEC 23008-2. The data must include the start codes. We support 4:2:0 in Main, Main Still Picture, and Main10 profiles in resolutions up to 4096x2304 pixels if it is supported by decoding hardware. We recommend that a I-Frame interval of between one and two seconds; it is however very important that you recall that the SDK will inform you when I-Frame insertions are required. The recommended bit-rate of the stream is provided by `NDIlib_send_get_target_bit_rate`; for HX it is considered reasonable that you allow the user to select whether you wish a bit-rate in the range of 0.1 – 2.0x the bit-rate provided by this function.

The recommended rules for the H.265 frame types are the same as those for the H.264 frame types described in the above section.

23.6.4.2 EXTRA DATA

The extra ancillary data required to configure an H.265 codec must correctly be provided. This must exist for all I-frames. The structure of this data should contain concatenated NAL units in Annex B format, along with their start codes. For H.265, they are the VPS, SPS, and PPS NAL units.

23.6.4.3 PTS AND DTS

The PTS and DTS are not used directly by the NDI SDK, however they are important in order to ensure that frames may be decoded and displayed in correct order. While we recommend that you use 100 ns intervals for these values, any time-base is technically supported as long as the ordering of integers is correct.

23.6.5 AAC SUPPORT

23.6.5.1 SUPPORTED FORMATS

AAC audio submitted to the NDI SDK should be in the raw AAC format.

All reasonable AAC bitrates are supported, although it is recommended that you use 192 kbps as a reasonable compromise. Any valid AAC channel count and sample-rate should be supported.

23.6.5.2 EXTRA DATA

The extra ancillary data required to configure an AAC codec must correctly be provided. This must exist for all AAC frames. The structure of this data should be the `AudioSpecificConfig` structure followed by `GASpecificConfig` as specified from ISO/IEC 14496-03. In practice, this means the extra-data section should be two bytes for each audio frame that describes the sample rate, channel count, etc. The information set in this structure should correctly match the audio settings header on the same frame.

23.6.5.3 AUDIO LEVELS

The AAC audio level matches that of the NDI SDK, meaning that a floating-point sine-wave value of 1.0 represents a +4 dBU equivalent audio level.

23.6.6 OPUS SUPPORT

Starting in version 5 of the NDI SDK one may transmit Opus audio through NDI. All audio is in the multi-channel Opus format, allowing up to 255 channels of audio within a packet. The FourCC used should be be `NDIlib_FourCC_audio_type_ex_OPUS`,

The properties provided within `NDIlib_audio_frame_v3_t` must reflect all legal values for the Opus codec, meaning that all bitrates, channel counts and valid packet sizes are supported. All audio within NDI uses the floating-point decompression functions, meaning that clipping would not occur if full range audio streams and it is recommended that you use the floating point versions of the compression. The Opus audio level matches that of the NDI SDK, meaning that a floating-point sine-wave value of 1.0 represents a +4 dBU equivalent audio level.

Unlike other compressed formats, the raw audio data is provided within the `NDIlib_audio_frame_v3_t` structure data fiels and a `NDIlib_compressed_packet_t` is not used. This distinction is made to reduce the bandwidth required for Opus audio over WAN networks by removing the need for extra PTS, DTS values which are not required for the decompression of the Opus bit-stream.

23.6.7 LATENCY OF COMPRESSED STREAMS

It is very important that you look very closely at your streams and ensure that they are configured for low latency decoding which is non-trivial and many “off the shelf” H.264 and HEVC encoders do not provide without changing their settings. If an encoder has a low latency mode then this is often a very good starting point. Other areas to look at are ensuring that you are using only I and P frames so that each incoming frame can be immediately decoded without a delay. In addition, it is very common that different codecs place “video delay” settings in the SPS NAL or SEI NAL units in the stream that instruct a decoder to delay the display of frames which introduces latency. In our experience, almost no HX implementation has been correctly configured for lowest latency use by default and work as always been needed to optimize the stream settings.

We have tools that can help measure the decode latency imposed by the stream if you wish.

23.6.8 STREAM VALIDATION

In an attempt to help highlight some potential problems, the NDI HX SDK will validate many aspects of streams that are passed to it. It is designed so that it will display the associated error on STDOUT if your stream is incorrect and then terminate the stream (to highlight that it is not a valid stream).

Obviously, this can make it quite important to monitor STDOUT in your implementation.

Important Note: To ensure that NDI HX works correctly across all devices and that a great end-user experience can be expected, it is very important that you take significant time testing to confirm that your implementation fully complies with this document and works well in practice.

23.7 EXTERNAL TALLY SUPPORT

Some sending might have knowledge of whether they are on output that might come from another non-NDI device. In effect, the sender side wants to indicate how it is known to be used and have that be reflected both to the local device but also down-stream in the "echoed tally messages". In order to achieve this there is a function on the sender side that updates the tally state.

```
bool NDIlib_send_set_tally(NDIlib_send_instance_t p_instance, const NDIlib_tally_t*
p_tally);
```

This function will "reference" count the local tally and the remote tally to give an indicator both on the local device and also on the remote connections on the combined NDI and local tally values.

23.8 KVM SUPPORT

The NDI advanced SDK includes supper for controlling a remote KVM enabled device over NDI. This SDK allows you to send keyboard, mouse, clipboard, and touch messages to a sender that is marked as supporting these messages. This functionality is implemented on an NDI receiver and messages are sent from this to a sender to offer KVM control.

It is critical that you test any applications you have very thoroughly since full support of a remote machine is not simple and has many edge cases that are hard to ensure solve – for instance for every keyboard and mouse "press" there must also be a corresponding "release message" or the remote machine will interpret this as a "stuck key".

The following documentation describes all of the relevant functions required for KVM control.

```
bool NDIlib_recv_kvm_is_supported(NDIlib_recv_instance_t p_instance);
```

This function will tell you whether the current source that this receive is connected to is able to be KVM controlled. The value returned by this function might change based on whether the remote source currently has KVM enabled or not. If you are receiving data from the video source using `NDIlib_recv_capture_v2`, then you will be notified of a potential change of KVM status when that function returns `NDIlib_frame_type_status_change`.

```
bool NDIlib_recv_kvm_send_left_mouse_click(NDIlib_recv_instance_t p_instance);
bool NDIlib_recv_kvm_send_middle_mouse_click(NDIlib_recv_instance_t p_instance);
bool NDIlib_recv_kvm_send_right_mouse_click(NDIlib_recv_instance_t p_instance);
```

When you want to send a mouse down (i.e. a mouse click has been started) message to the source that this receiver is connected too then you would call this function. There are individual functions to allow you to send a mouse click message for the left, middle and right mouse buttons. To simulate a double-click operation you would send three messages; the first two would be mouse click messages with the third being the mouse release message. It is important that for each click message there must be at least one mouse release message or the operating system that you are connected too will continue to believe that the mouse button remains pressed forever.

```
bool NDIlib_recv_kvm_send_left_mouse_release(NDIlib_recv_instance_t p_instance);
bool NDIlib_recv_kvm_send_middle_mouse_release(NDIlib_recv_instance_t p_instance);
bool NDIlib_recv_kvm_send_right_mouse_release(NDIlib_recv_instance_t p_instance);
```

When you wish to send a mouse release message to a source, then you will call one of these functions. They simulate the release of a mouse message and would always be sent after a mouse click message has been sent.

```
bool NDIlib_recv_kvm_send_vertical_mouse_wheel(NDIlib_recv_instance_t p_instance, const
float no_units);
bool NDIlib_recv_kvm_send_horizontal_mouse_wheel(NDIlib_recv_instance_t p_instance, const
float no_units);
```

To simulate a mouse-wheel update you will call this function. You may simulate either a vertical or a horizontal wheel update which are separate controls on most platforms. The floating-point unit represents the number of "units" that you wish the mouse-wheel to be moved, with 1.0 representing a downwards or right-hand single unit.

```
bool NDIlib_recv_kvm_send_mouse_position(NDIlib_recv_instance_t p_instance, const float
posn[2]);
```

To set the mouse cursor position you will call this this function. The coordinates are specified in resolution independant settings in the range 0.0 - 1.0 for the current display that you are connected too. The resolution of the screen can be known if you are receiving the video source from that device and this might change on the fly of course. Position (0,0) is the top left of the screen. posn[0] is the x-coordinate of the mouse cursor, posn[1] is the x-coordinate of the mouse cursor.

```
bool NDIlib_recv_kvm_send_clipboard_contents(NDIlib_recv_instance_t p_instance, const char*
p_clipboard_contents);
```

In order to send a new "clipboard buffer" to the destination one would call this function and pass a null terminated string that represents the text to be placed on the destination machine buffer.

```
bool NDIlib_recv_kvm_send_touch_positions(NDIlib_recv_instance_t p_instance, const int
no_posns, const float posns[]);
```

This function will send a touch event to the source that this receiver is connected too. There can be any number of simultanous touch points in an array that is transmitted. The value of no_posns represents how many current touch points are used, and the array of posns[] is a list of the [x,y] coordinates in floating point of the touch positions. These positions are processed at a higher precision that the screen resolution on

many systems, with (0,0) being the top-left corner of the screen and (1,1) being the bottom-right corner of the screen. As an example, if there are two touch positions, then posns[0] would be the x-coordinate of the first position, posns[1] would be the y-coordinate of the first position, posns[2] would be the x-coordinate of the second position, posns[2] would be the y-coordinate of the second position.

```
bool NDIlib_recv_kvm_send_keyboard_press(NDIlib_recv_instance_t p_instance, const int
key_sym_value);
```

In order to send a keyboard press event, this function is called. Keyboard messages use the standard X-Windows Keysym values. A copy of the standard define that is used for these is included in the NDI SDK for your convenience with the filename "Processing.NDI.KVM.keysymdef.h". Since this file includes many #defines, it is not included by default when you simply include the NDI SDK files, if you wish it to be included you may #define the value NDI_KVM_INCLUDE_KEYSYM or simply include this file into your application manually. For every key press event it is important to also send a keyboard release event, or the destination will believe that there is a "stuck key"! Additional information about keysym values may easily be located online, for instance https://www.tcl.tk/man/tcl/TkCmd/keysyms.html

```
bool NDIlib_recv_kvm_send_keyboard_release(NDIlib_recv_instance_t p_instance, const int
key_sym_value);
```

Once you wish a keyboard value to be "release" then one simply sends the matching keyboard release message with the associated key-sym value.

23.9 NDI ADVANCED SDK EXAMPLE DESIGN

The NDI Advanced SDK contains working example designs intended to assist in using the Advanced SDK NDI software SDK and the FPGA NDI IP cores. Building a working Advanced SDK NDI design requires wide-ranging expertise and these example designs are provided in an attempt to make this process more accessible.

Pre-built uSD card images are available for the supported platforms, and details are provided regarding how to rebuild each required piece from scratch.

23.9.1 QUICKSTART

1. Obtain one of the supported development boards:

   - Digilent Zybo-Z7-20
   - Terasic SoCKit + DVI-HSMC (optional)
   - Xilinx ZCU104

2. Boot your board using a pre-built image.

See Section 23.14, “uSD Images for NDI demo platforms” for details on obtaining the uSD image, writing it to a uSD card, and using it with one of the supported platforms.

23.9.2 DIRECTORY STRUCTURE:

Filename	Contents
README.md	High level overview of the provided example projects
README.uSD.md	Details on using the pre-built uSD images
CHANGELOG.md	List of notable changes
src/cpp	Example software project files
src/fpga	Hardware design files and example projects
linux_kernel	Projects to build kernel and boot loader
os_uSD	Scripts to generate root filesystem and bootable uSD images

23.9.3 DESIGN OVERVIEW

There are a lot of different pieces required to create a working example system:

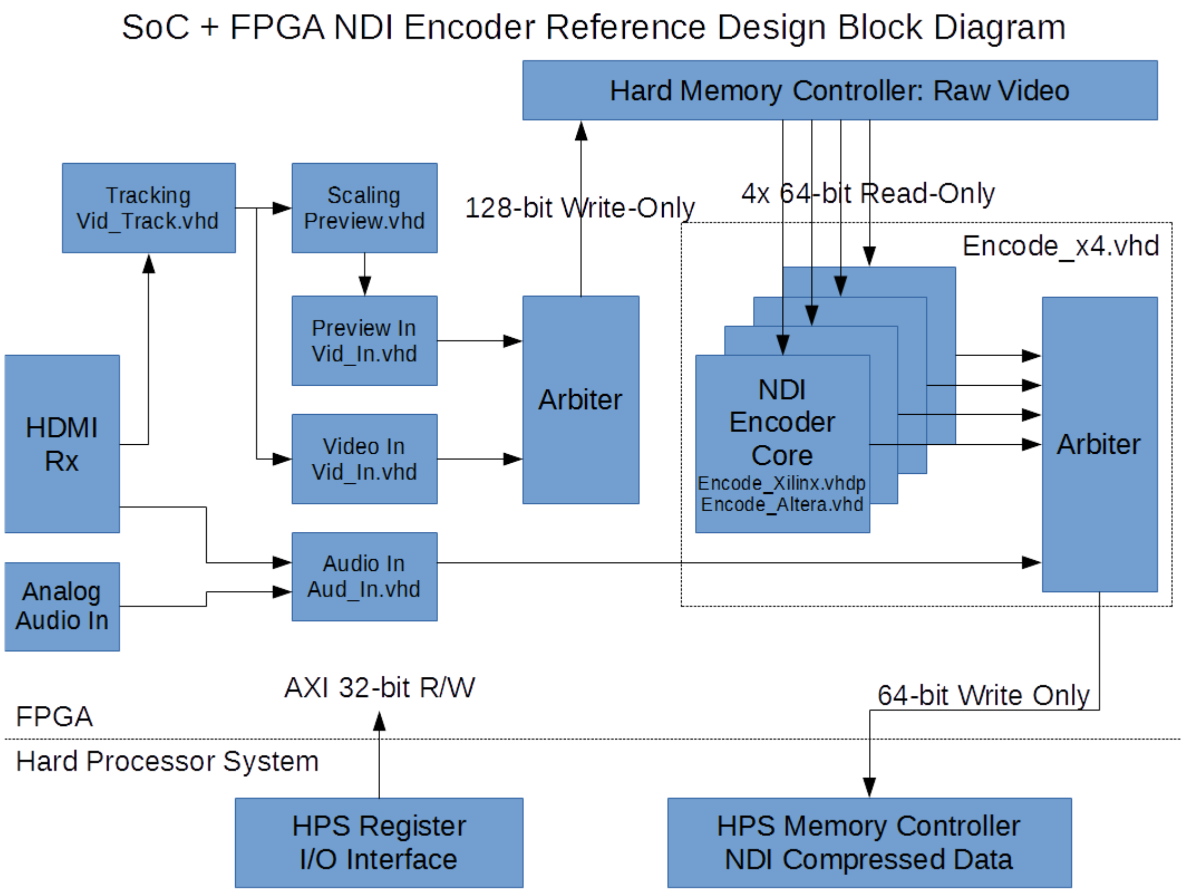
- FPGA hardware design
- Boot loader
- Linux kernel
- Device-tree
- Linux root file system
- Application software
- Bootable image including all of the above

While each of these components is important and necessary, not all of them have to be customized in order to create a custom project.

The three components that need to be customized for almost any project are the actual FPGA hardware, the application software, and the device tree. The Linux kernel and boot loader can typically be used without customizations and there are several options for creating a Linux root file system.

This example design uses the Xilinx PetaLinux distribution for the boot loader, the Linux kernel, and most of the device tree. The root file system is a standard Debian install. The FPGA logic and application software are custom, and some device-tree customizations are required to support the custom FPGA hardware.

Each of these components is discussed in more detail below.



The resource usage of the 4-core version of the NDI encoder which can encode up to 4Kp60 video is listed below:

Xilinx		Altera	
Target Part	XC7Z020-1CLG400C	Target Part	5CSEMA2U23C7
FMax(NDI)	200 MHz	FMax(NDI)	200 MHz
FMax(Reg)	100 MHz	FMax(Reg)	100 MHz
Total LUTs	7270	ALM(used)	7159.6
Logic LUTs	7250	ALM(needed)	5816.3
SRLs	20	ALUTs	7058
FFs	11833	FFs	13140
RAMB36 / RAMB18	12 / 12	M10Ks	56
DSP48	36	DSPs	20

23.9.4 THEORY OF OPERATION

Advanced SDK systems require interaction between hardware and software. This is a complex process even on small micro-controllers, and is even more so on systems running a high-level OS such as Linux.

This example design was created with the intent to make the interaction between hardware and software as simple as possible to implement and modify, with no need to write kernel-space code.

23.9.4.1 COMMON ENCODE AND DECODE BASE FUNCTIONALITY

A. LOW LEVEL DETAILS AND DEVICE-TREE

In Advanced SDK systems, it is necessary to communicate details of the system across several fundamentally different environments (e.g., hardware, the Linux kernel, and application software). Rather than using lots of "magic" numbers stashed in cryptic header files, this is now mostly done at the system level with device-tree. Details provided via device-tree can control which kernel modules get loaded (or do not), as well as modify the behavior of those modules.

A.1 HARDWARE IP

- Xilinx IP

Standard Xilinx IP blocks for GPIO and I2C are implemented along with the Zynq PS core in the Xilinx block design files.

These IP blocks are automatically added to the device-tree by the PetaLinux tools, and the kernel drivers for these IP blocks are automatically enabled. This allows the pushbutton switches and LEDs connected to the PL fabric to be made available to the Linux kernel as part of the standard gpio subsystem.

- Xilinx HDMI IP

The ZCU104 uses the Xilinx HDMI IP core, however the Linux drivers for this core are disabled since the NDI encoding data flow operates outside of the Linux kernel's standard video input and output systems. The control software for the HDMI core instead operates on one of the Cortex-R5 cores, which means all the hardware used by the HDMI core (uart1, i2c0, i2c1) must be disabled in Linux to avoid contention.

- Custom IP

The NDI hardware implements a 64K block of register space which is manually added to the device-tree along with IRQ settings and other hardware details. The generic-uio driver is used to map the hardware register space and IRQs so they are available to user-space code.

A.2 RESERVED MEMORY REGIONS

This design requires two large blocks of buffer memory, one for compressed video and raw audio data that must be visible to Linux, and one for raw video data which can be visible to Linux but may also be implemented as a separate memory bank for performance if necessary.

- NewTek_Reserved

This memory region is used to store compressed video data and raw audio data. This region is marked cacheable for performance. Cache coherency is maintained since the hardware accesses this memory region using a cache coherent bus (S_AXI_ACP on the Zynq-7000 and S_AXI_HPC on the Zynq Ultrascale+).

- NewTek_Video

This memory region is used to store raw (uncompressed) video data, and thus requires high bandwidth. The application does not access this memory in normal operation so this region may be implemented as a dedicated FPGA-side memory bank to improve performance.

This region will only appear in the device-tree if implemented using shared memory visible to Linux. If a live video input is not available and this memory region is visible to Linux, a video pattern can be written to allow testing of the compression core and software application.

A.3 SOFTWARE

- Standard Xilinx IP Blocks

The standard Xilinx IP blocks (axi_gpio, axi_iic) are automatically added to the device tree and stock Linux kernel drivers are used to communicate with this hardware.

- Custom IP blocks

The device-tree entries for the custom FPGA IP use the generic-uio kernel driver. This driver makes the memory regions and IRQs specified by the device-tree entries accessible to user-space code. The libuio library is used to facilitate accessing the hardware.

The application also reads details regarding the reserved memory regions directly from the device-tree, as this functionality is not supported by libuio.

B. INITIAL STARTUP

File(s)	Description
<device-tree>	Contains memory addresses and IRQ details
src/cpp/ndi_common/hardware.*	Low-level access to register space and IRQs
src/cpp/ndi_common/device.*	Machine specific details

src/fpga/ip/common/Version.vhd	Version information compiled into the hardware
src/cpp/ndi_encode/ndi_encode.*	Top level application file (Encode)
src/cpp/ndi_decode/ndi_decode.*	Top level application file (Decode)

When the application is launched, quite a bit of setup is performed before continuing operation is passed to a number of created threads.

Most of this setup happens when the hardware class is initialized. After the software processes any command-line options, an instance of the hardware class is created and initialized.

The hardware class uses libuio to access the memory regions and interrupts specified in the device-tree for the custom hardware IP. In addition, details for the reserved memory regions are read from the device-tree and the memory is mmap()'d to make it available for access.

At this point, version information is read from the hardware and a device class specific to the hardware platform is created. This allows some behaviors to change between different physical boards (currently used to control any required ACP address translation).

Finally, details regarding the UIO devices and reserved memory are printed to the console, and the reserved memory region for compressed video is written with the value 0xdeaddead, making it easier to identify uninitialized memory locations when debugging.

23.9.4.2 NDI ENCODE OPERATION

A. VIDEO INPUT

File(s)	Description
src/cpp/ndi_encode/video_capture.*	video_capture:: class
src/cpp/ndi_encode/track.*	track:: class
src/fpga/ip/common/Vid_Track.vhd	video format tracking hardware
src/fpga/ip/common/Vid_In.vhd	Video input DMA logic
src/fpga/ip/common/Preview.vhd	Creates low-resolution preview stream
Filename varies	Low level video input logic (eg: HDMI or SDI)

Video input starts with the low-level video input logic. This logic is platform specific with the ZCU104 currently using the Xilinx HDMI IP core while the Zybo-Z7 uses an open-source Arctix-7 HDMI implementation modified as needed.

The software for controlling the low-level video input logic is currently outside the scope of this example. The Zybo-Z7 HDMI input requires no control software, and the Xilinx HDMI control software is running bare-metal on one of the Cortex-R5 cores (see the README file accompanying the Vivado hardware project files for details).

There is, however, software support for low-level input monitoring code to send details (locked/unlocked, resolution, etc) to the video_capture:: class. See video_capture::set_signal() and the do_commands() function in ndi_send.cpp for details.

Once the low-level video signal is received by the FPGA hardware, the signal is synchronized to the main clock and converted to a standard interface (HDMI_T). This signal is sent to the format tracking logic (Vid_Track.vhd) and is filtered (Preview.vhd) to create a low resolution preview stream. The resulting full and preview resolution streams are each sent to a bus mastering DMA engine (Vid_In.vhd) which assembles pixels into words and writes the raw video data into memory.

The track:: class monitors the video format tracking hardware to detect the acquisition or loss of signal lock. When either event happens, appropriate details are communicated to the video_capture:: class to start or stop video streaming.

The video_capture:: class manages the recording of video data into the reserved memory buffer as well as hardware settings controlling things like the data format and preview decimation ratio.

Since there are two input streams, the video_capture:: class operates on frame pairs. Each pair always contains a full resolution frame and may or may not contain a preview resolution frame. The maximum preview framerate is 30 fps, so if the full framerate is greater than 30 fps, some full frames will not have a corresponding preview frame. For simplicity, this is represented as a frame pair with an invalid preview pointer (`NULL`).

The video_capture::capture_frames thread loops through the following process to capture frame pairs:

- Create a new frame pair
- Allocate memory from the reserved buffer
- Fill in frame details
- Queue the frame (send details to the hardware)
- Post the frame (tell hardware the new frame data is valid)
- Wait for an interrupt indicating the frame has been captured
- Send the frame to the compression logic

Once a frame pair is captured, it is placed on a work queue for the video compression engine. If this queue is too full, the oldest pending frame pair is dropped and a warning is printed.

NOTE: The video input logic assumes a "clean" video signal and is an example design intended to be simple and easy to understand. This logic does not attempt to deal with real world problems like random noise on an unterminated SDI input or when users plug/unplug the cable while the system is running. An actual production device should be more tolerant of signal errors.

B. VIDEO PATTERN

File(s)	Description

src/cpp/ndi_encode/video_pattern.*	video_pattern:: class

The video_pattern:: class is a sub-class of the video_capture:: class that does not rely on hardware to generate a video stream. Rather than capturing frames from hardware, this class generates a video pattern any time the signal format changes (it is possible to manually change the resolution at run time from the console).

The video pattern generated is twice as tall as the video format (see the m_scroll_dist member variable), and a preview resolution version of the pattern is also created (since both full and preview resolution streams are required). In normal operation, a frame-sized window of the over-sized video pattern is used to provide source video for the video_compress:: class, with the start point of the window moving down one line with each new frame, providing the illusion of moving video.

C. VIDEO COMPRESSION

File(s)	Description
src/cpp/ndi_encode/video_compress.*	video_compress:: class
src/fpga/NDI_Enc/Encode_x4.vhd	4-core NDI Hardware Encoder
src/fpga/NDI_Enc/Encode_Xilinx.vhdp	Single NDI Hardware Encoder core

One, two, or four copies of the NDI Encoder core (Encode_Xilinx.vhdp) are instantiated in hardware (Encode_x4.vhd). The cores operate in parallel, with each core operating on one or more quarters of the video frame known as a "slice". This increases the effective throughput of the compression core and lowers system latency.

The video_compress:: class monitors a work queue filled with raw video frames by the video_capture:: class (above). When a new frame pair is received the full-resolution frame is sent to hardware for processing. Once the hardware generates an interrupt indicating processing is finished, the software performs minimal post-processing on the compression thread. Slice lengths are read from the hardware and written to memory by fixup_frame(), the resulting frame length is used to update the quality setting, then the frame is passed to the copy thread via add_frame_copy().

The copy thread monitors a work queue which is filled with compressed frames by the compression thread. When a new frame arrives, the copy thread allocates a contiguous chunk of memory and copies the compressed slice data generated by the hardware compression engine into it. Once copied, the frame is passed to one of the sending threads via add_frame_ndi(). The memory copy is a fairly expensive operation and has been given it's own thread so the time required to copy the data will not limit the processing time available to the NDI sending threads.

The send threads (send_full, send_prvw) monitor independent work queues for the full and preview resolution video streams and pass new frames to the NDI stack. Sending a frame to the NDI stack is a synchronizing event which will block until the previously frame is fully processed, which is why there are independent threads for the two video streams. This gives each stream the maximum amount of time to send data to the listeners.

D. VIDEO TRANSMISSION

File(s)	Description
src/cpp/ndi_encode/video_compress.*	video_compress:: class
src/cpp/ndi_encode/network_send_video.cpp	network_send:: video support

The send_full() and send_prvw() threads in the video_compress:: class copy the four slices of compressed data generated by the hardware into a single contiguous frame and pass it to network_send::add_frame(). This function is basically a shim which converts the video_compress::frame_t type into an NDIlib_video_frame_v2_t type and sends it to the NDI stack.

NOTE: The various frame types exist primarily because much of the example code predates the availability of the official NDI Advanced SDK. Expect future releases of this code to migrate to using native NDI types.

E. AUDIO INPUT

File(s)	Description
src/cpp/ndi_encode/audio_capture.*	audio_capture:: class
src/fpga/ip/common/Aud_In.vhd	Audio input DMA logic

Video input starts with the low-level audio input logic. This logic supports standard iis serial audio streams. Support for parallel audio data extracted from the HDMI streams will be supported soon. Once assembled into complete audio samples, the data is queued in a FIFO allowing burst writes to system memory.

The audio_capture:: class manages the recording of audio data into the reserved memory buffer as well as hardware settings controlling things like the data format (typically left-justified). The audio_capture::capture_frames thread loops through a process very similar to the video_capture thread. One major difference is the audio logic currently supports "overlapped" commands, meaning the next command is written to hardware while the current command is still in progress:

- Create a new frame
- Allocate memory from the reserved buffer
- Fill in frame details
- Queue the frame (send details to the hardware)
- Post the frame (tell hardware the new frame data is valid)
- Loop until signaled to exit
- Create a new frame
- Allocate memory from the reserved buffer
- Fill in frame details
- Queue the frame (send details to the hardware)
- Post the frame (tell hardware the new frame data is valid)

- Wait for an interrupt indicating the frame has been captured
- Send the frame to the compression logic

F. AUDIO COMPRESSION

File(s)	Description
src/cpp/ndi_encode/audio_compress.*	audio_compress:: class

This class simply passes captured audio frames from the audio_capture class to the NDI stack. It exists primarily to match the video path data flow, to allow any processing of the audio samples that might be required (eg: masking lower-order bits that might contain AES packet data), and because when this code was initially written the NDI API did not support 32-bit signed audio samples.

In future versions, expect the native NDI types to be used and the audio_compress class to be removed.

G. AUDIO TRANSMISSION

File(s)	Description
src/cpp/ndi_encode/network_send_audio.cpp	network_send:: audio support

This function is basically a shim which converts the audio_capture::frame_t type into an NDIlib_audio_frame_interleaved_32s_t type and sends it to the NDI stack.

Expect this code to be deprecated in future versions and the sending logic to be migrated to the audio_capture:: class.

H. TALLY OPERATION

File(s)	Description
src/cpp/ndi_encode/tally.*	tally:: class

Tally outputs are implemented using the Linux kernel LED class. Tally LED entries are created in the device-tree, and the application uses the resulting sysfs entries to control the LED behavior. The tally class initializes all LEDs to a known state on construction, then the tally process simply loops, checking for updates from the NDI stack. If new tally data is available, the tally LEDs are updated.

23.9.4.3 REBUILDING FROM SOURCE

Directory	Description
src/fpga/	FPGA Project Files
src/fpga/	C++ Application Project Files

linux_kernel/	Linux kernel, U-Boot, and device-tree
os_uSD/	Debian rootfs and uSD images

A. FPGA HARDWARE DESIGN

The `src/fpga/` directory contains projects to build a bit-file for each supported platform. Refer to Section 23.10, "NDI® FPGA IP Core Example Designs" for full details on building the hardware projects.

B. BOOT LOADER, LINUX KERNEL, AND DEVICE-TREE

The `linux_kernel/` directory contains projects to build a boot-loader, kernel, and device-tree for the supported platforms. The device-tree customizations required for the NDI FPGA hardware are also included in the example projects. Refer to Section 23.12, "PetaLinux Projects for NDI® Xilinx hardware platforms" for full details.

B.1 PETALINUX ROOT FILE SYSTEM

NOTE: The scripts in `linux_kernel/` create a Petalinux root file-system in addition to the kernel and boot loader. However, this is a very minimal system, and it is difficult to customize since the entire OS is cross-compiled. This example uses a standard Debian OS install, which allows for self-hosted compiles and easy modification of the OS using standard package management tools.

For details on building the Debian root file-system, see the section on creating a bootable image (below).

C. APPLICATION SOFTWARE

The `src/cpp/` directory contains example encode and decode software applications which use the FPGA based NDI logic to (de)compress live video data, and the Advanced SDK NDI SDK to send and receive that data as an NDI stream. Refer to the Section 23.11, "NDI Advanced Application Examples" for details.

D. BOOTABLE IMAGE INCLUDING ALL OF THE ABOVE

The `os_uSD/` directory contains scripts to create a Debian root filesystem as well as a bootable uSD card image. The resulting images contain all prerequisites needed to perform a self-hosted build of the example NDI application (ie: built on the ARM platform and not cross-compiled). Refer to the Section 23.13, "uSD Image Builder" for full details.

23.10 NDI® FPGA IP CORE EXAMPLE DESIGNS

23.10.1 DIRECTORY STRUCTURE:

Directory	Contents
NDI_Dec/	VHDL files for the NDI® Decoder core. All files should be compiled into the NDI_Dec VHDL library rather than the default work library
NDI_Enc/	VHDL files for the NDI Encoder core. All files should be compiled into the NDI_Enc VHDL library rather than the default work library

ip/altera	Altera specific implementations of memory and FIFO blocks
ip/common	Source code common to all example designs
ip/extern	External (non-NewTek) IP used in the example designs
ip/packages	VHDL Packages defining various types used in the logic
ip/xilinx	Xilinx specific implementations of memory and FIFO blocks
SoCKit-Dec/	Quartus 19.1 NDI Decode project directory targeting the Terasic SoCKit
ZCU104/	Vivado 2019.2 NDI Encode project directory targeting the Xilinx ZCU104
ZCU104-Dec/	Vivado 2019.2 NDI Decode project directory targeting the Xilinx ZCU104
Zybo-Z7-20/	Vivado 2019.2 NDI Encode project directory targeting the Digilent Zybo-Z7-20
Zybo-Z7-20-Dec/	Vivado 2019.2 NDI Decode project directory targeting the Digilent Zybo-Z7-20
Zybo-Z7-20-Lite/	"Lightweight" Vivado 2019.2 NDI Encode project directory targeting the Digilent Zybo-Z7-20 with 16-bit SDRAM interface

23.10.2 BUILD DEPENDENCIES

23.10.2.1 XILINX

All Xilinx projects require Vivado version 2019.2. Any edition of Vivado 2019.2 will work, including the "WebPACK" edition available via free download from Xilinx.

A. ZCU104

You must have a valid license for the Xilinx HDMI IP core. You may use a full license or a free evaluation license, but the license must be installed prior to launching Vivado to build the project:

https://www.xilinx.com/products/intellectual-property/hdmi.html

B. ZYBO-Z7-20 / ZYBO-Z7-20-LITE

Board files for the Zybo-Z7-20 must be installed from Digilent in order to build the Zybo example designs. The board files may be obtained from github: https://github.com/Digilent/vivado-boards/ and installation instructions are on the Digilent website: https://reference.digilentinc.com/reference/software/vivado/board-files

23.10.2.2 INTEL/ALTERA

All Intel/Altera projects require Quartus 19.1 and the SoC FPGA EDS. The examples were compiled using the free "Lite" version, however the "Standard" edition should work as well. Follow the Altera instructions for installation, including the required manual installation of Cygwin if you are using Windows.

The contents of the file `NDI_Enc/Encode_Altera.lic` (for Encode) and `NDI_Dec/Decode_Altera.lic` (for Decode) must be appended to your Quartus license file. If you do not have a license file, you must create one.

IMPORTANT: If you are using Windows 10, make sure you download and install the patch "Intel® Quartus® Prime 19.1 Patch 0.02std for Windows" per KDB article 14010725090.

23.10.3 REBUILDING

23.10.3.1 XILINX

Once you have all required dependencies installed (see above), simply open the desired project and build normally. The contents for the block design elements (hard processor system, PLLs, etc) will be regenerated on the first build.

23.10.3.2 ALTERA

Before you can build the Altera projects, you must generate the hps platform logic from the qsys file.

A. REBUILD THE HPS PLATFORM

- Launch Platform Designer
- Open ndi_hps.qsys
- Click "Generate HDL..."
- Select VHDL Synthesis output and click "Generate"

NOTE: If the fpga_sdram sequencer generation fails, verify you have *EXACTLY* followed the Altera installation instructions for Quartus, WSL, Cygwin, and the required patch for Windows. Alternately, you can use a Windows 7 or Linux based Quartus install instead.

- Once the qsys project has been generated, run a normal Quartus compilation

The following warnings are expected and harmless:

```
Warning: ndi_hps.fpga_sdram: The 'Type' value must be set to Bidirectional if you plan to
simulate the example design.
Warning: ndi_hps.fpga_sdram: 'Quick' simulation modes are NOT timing accurate. Some
simulation memory models may issue warnings or errors
Warning: ndi_hps.fpga_sdram.pll_bridge: pll_bridge.pll_sharing cannot be both connected and
exported
Warning: ndi_hps.fpga_sdram: The 'Type' value must be set to Bidirectional if you plan to
simulate the example design.
Warning: ndi_hps.fpga_sdram: 'Quick' simulation modes are NOT timing accurate. Some
simulation memory models may issue warnings or errors
Warning: ndi_hps.fpga_sdram.pll_bridge: pll_bridge.pll_sharing cannot be both connected and
exported
Warning: hps_0.f2h_irq0: Cannot connect clock for irq_mapper.sender
Warning: hps_0.f2h_irq0: Cannot connect reset for irq_mapper.sender
Warning: hps_0.f2h_irq1: Cannot connect clock for irq_mapper_001.sender
Warning: hps_0.f2h_irq1: Cannot connect reset for irq_mapper_001.sender
```

23.10.4 KNOWN ISSUES AND LIMITATIONS:

- The NDI Encode core for Altera is proven in hardware, but no example project is currently available (coming soon!)
- The Altera NDI Decode project compiles and runs, but does not meet timing constraints, has not been fully tested, and should be considered beta.

23.10.5 NDI IP FILES AND USE

23.10.5.1 NDI ENCODER

Two VHDL files are required to use the NDI Encoder. Both files should be compiled into the NDI_Enc library in the following order:

A. XILINX PROJECTS:

```
Encode_Xilinx.vhdp
Encode_x4.vhd
```

B. ALTERA PROJECTS:

```
Encode_Altera.vhd
Encode_x4.vhd
```

There is also a component declaration file `Enc_Core_Comp.vhd` for use as a reference when instantiating the encoder core. Each encoder core `Enc_Core_E` includes a register I/O interface, a read-only bus master interface for reading raw video data, and a write-only bus master interface for writing compressed NDI data.

Note the raw video memory and the compressed NDI memory do not need to be the same physical bank of memory. Using two memory banks can improve system performance, particularly on lower-end devices such as the Zynq 7000 and Cyclone-V families.

23.10.5.2 NDI DECODER

Two VHDL files are required to use the NDI Decoder. Both files should be compiled into the NDI_Dec library in the following order:

A. XILINX PROJECTS:

```
Decode_Xilinx.vhdp
Decode_x4.vhd
```

B. ALTERA PROJECTS:

```
Decode_Altera.vhd
Decode_x4.vhd
```

23.10.5.3 SDRAM BUS INTERFACES

The bus-mastering memory interfaces are natively a sub-set of the Altera Avalon interface specification. These buses may be tied directly to an Avalon interface port in an Altera design. For Xilinx designs, clear-text bus shim logic is provided to convert the Avalon bus subset used by the Encoder core into a more standard AXI interface:

File	Function

Avl_Axi_Rd.vhd	Translates Avalon read bus to AXI read bus
Avl_Axi_Wr.vhd	Translates Avalon write bus to AXI write bus
Avl2_Axi_rd.vhd	Merges 2 Avalon read buses into one AXI read bus
AXI_Reg_Wr.vhd	Interface AXI write bus to simple register write bus
AXI_Reg_Rd.vhd	Interface AXI read bus to simple register read bus

Usage examples for these files can be found in the top-level files for the example designs.

A. NDI ENCODE

A.1 WRITES (COMPRESSED NDI DATA)

There is clear-text logic in the Encode_x4 file which merges the four lower bandwidth compressed write streams into a single write stream. The raw (uncompressed) audio data is also merged into this stream.

This write bus is connected to a cache coherent port to the hard processor system, eliminating the need for a kernel mode DMA driver.

A.2 READS (RAW VIDEO DATA)

For maximum performance, the four read streams are all brought out to the top level and are tied to four dedicated ports on the hard memory controller (Zybo-Z7) or are merged and connected to two dedicated ports (ZCU104). This is to allow the SDRAM controller more outstanding transactions which generally improves overall throughput. If necessary, the encoder read ports may be merged into fewer streams, although this may affect performance.

B. NDI DECODE

B.1 READS (COMPRESSED NDI DATA)

There is clear-text logic in the Decode_x4 file which merges the four lower bandwidth compressed Read streams into a single AXI read stream. This read bus is connected to a cache coherent port to the hard processor system, eliminating the need for a kernel mode DMA driver.

B.2 WRITES (RAW VIDEO DATA)

For maximum performance, the four write streams are all brought out to the top level and are tied to four dedicated ports on the hard memory controller. This is to allow the SDRAM controller more outstanding transactions which generally improves overall throughput. If necessary, the decoder write ports may be merged into fewer streams, although this may affect performance.

23.10.5.4 FULL VS. LITE ENCODE DESIGNS

The `ZCU104` and `Zybo-Z7-20` example designs are a "full" implementation of the NDI encoder and instantiate four encoder cores.

The `Zybo-Z7-20-Lite` example design is intended as a reference for a more limited platform. Only one SDRAM chip is used (16-bit SDRAM interface) and only two encoder cores are instantiated. This implementation is still capable of operation at resolutions up to 1080p60.

23.10.6 HDMI INPUT LOGIC

23.10.6.1 ZYBO-Z7-20 HDMI INPUT

The Zybo-Z7 board uses the ISERDESE2 and IDELAYE2 primitives to convert the serial HDMI data into parallel video. Two open-source cores are supported, each with its own advantages and disadvantages. Currently, the Digilent dvi2rgb core is used by default, however this can be changed by modifying the USE_DVI generic passed to the top-level logic during synthesis:

- Select `Project Manager -> Settings`
- Select `Project Settings -> General`
- Click the `...` button next to `Generics/Parameters`
- Edit the value for the `USE_DVI` generic as desired:
- true = Use the Digilent dvi2rgb core (default)
- false = Use the Hamsterworks HDMI core

A. DIGILENT DVI2RGB CORE

This core only supports DVI inputs, so no HDMI features are supported (eg: Data Island Packets including AVI InfoFrames and Advanced SDK audio).

The low-level serial-to-parallel conversion works very well, however, and automatically aligns each input lane to the center of the "eye" in the incoming bitstream, then insures each recovered parallel stream is aligned with the other two.

B. HAMSTERWORKS HDMI CORE

This core supports HDMI features including InfoFrame and Audio Sample Packets, but the low-level serial-to-parallel logic does not align to the center of the "eye" in the incoming bitstream. This results in occasional glitches in the video input stream, so the dvi2rgb core is currently used as the default.

B.1 ZCU104 HDMI INPUT

The ZCU104 board uses the Xilinx HDMI 2.0 IP core. A full or evaluation license must be installed for this core prior to building the design with Vivado. This IP core also requires control software to monitor the incoming HDMI signal and make any adjustments required. This software is currently running "bare-metal" on one of the Cortex-R5 cores.

C. REBUILDING THE CORTEX-R5 HDMI SOFTWARE

- Compile the ZCU104 FPGA design
- Export the hardware to the SDK
- Select `File -> Export -> Export Hardware...`
- Select `<Local to Project>` and click `OK`
- Launch the SDK `File -> Launch SDK`
- Exported location: `<Local to Project>`
- Workspace: `<Local to Project>`

- Click `OK`
- Disable automatic builds while we are creating a new BSP and Project
- Insure `Project -> Build Automatically` is not checked
- Create a new BSP `File -> New -> Board Support Package`
- Set Project name to `standalone_bsp_R5_0`
- Leave `Use default location` set
- Leave Hardware Platform set to `ZCU104_NDI_HDMI_hw_platform_0`
- Set CPU to `psu_cortexr5_0`
- Select `standalone` Board Support Package OS
- Click `Finish`
- The `Board Support Package Settings` window should open
- Select `Overview` if not already selected
- Select the `libmetal` and `openamp` libraries (used to communicate with Linux), leave all other libraries unselected
- Switch the console to UART1
- Select `Standalone`
- Set the Value for stdin to `psu_uart_1`
- Set the Value for stdout to `psu_uart_1`
- Click `OK`
- Be patient, the BSP generation can take a while.
- Import the HDMI Rx example project
- Open the system.mss file if it is not already open:
- `Project Exporer -> standalone_bsp_R5_0 -> system.mss`
- Scroll down to `v_hdmi_rx_ss` under `Peripherial Drivers` (it is near the bottom)
- Select `Import Examples`
- Select `RxOnly_R5`
- Click `OK`
- Be patient!
- Rename the project (Optional)
- Right-click `standalone_bsp_R5_0_RxOnly_R5_1`
- Select `Rename`
- Change name to `NewTek_HDMI_RxOnly_R5`
- All file paths below are relative to this directory!
- Update the DDR memory region in the linker script
- Open the file `src/lscript.ld`
- Set the `psu_r5_ddr_0_MEM_0` region details to match the rproc_0_reserved reserved memory region in the device-tree and save the file Base Address : 0x3ed00000 Size : 0x80000
- Change HDMI code to use UART1
- Open `src/xhdmi_example.h`
- At line 109, change the UART_BASEADDRESS to use UART1 and save the file #define UART_BASEADDR XPAR_XUARTPS_1_BASEADDR
- Update EDID Data (Optional)
- Open `src/xhdmi_edid.h`

- Edit constant `Edid[]` as desired
- Build the project `Project -> Build All`
- Re-enable automatic builds if desired: `Project -> Build Automatically`
- Copy the elf file to the ZCU104
- The elf file can be found at the following path
- `<sdkdir>/<projdir>/Debug/<projname>.elf`
- Copy this file to `/lib/firmware/` on the ZCU104
- Make sure you reference this filename when launching the R5 via remoteproc!

23.10.7 HDMI OUTPUT LOGIC

23.10.7.1 ZYBO-Z7-20 HDMI INPUT

The Zybo-Z7 board uses the DVI output logic from the "Hamsterworks" HDMI project (ip/extern/HDMI/src/dvid_output.vhd). This is a simple DVI output and no HDMI features (e.g., Advanced SDK audio) are currently supported. Both 74.25 MHz (720p) and 148.5 MHz (1080p60) pixel rates are supported.

23.11 NDI ADVANCED APPLICATION EXAMPLES

This is a set of example applications illustrating the use of the FPGA based NDI® encoder with the Advanced SDK NDI SDK. For full details and a theory of operation covering the entire system (software, hardware, and OS), refer to the top-level README file.

Source code for the examples is organized into the following subdirectories:

Directory	Contents
ndi_common/	Files common to all NDI applications
ndi_decode/	Example NDI Decoder application (eg: TV or Monitor)
ndi_encode/	Example NDI Encoder application (eg: Camera or NDI Source)
ndi_util/	Debug and utility applications

23.11.1 GETTING STARTED

23.11.1.1 PREREQUISITES

The following prerequisites must be installed before you can build the example applications.

A. LIBUIO

- https://github.com/Linutronix/libuio

libuio is used to access the FPGA hardware via entries in the device-tree

The available uSD card images have libuio packages compiled from source (see `~/libuio`) and the debian packages installed. For other platforms, libuio will need to be compiled and installed manually.

B. NDI ADVANCED SDK LIBRARY

The NDI Advanced SDK allows the sending and receiving of compressed frames, which is required to use the FPGA encoder.

To install the library, simply copy the files to an appropriate location for your system. The example uSD card images have the NDI library files already installed in `/usr/local/lib` and `/usr/local/include.`

23.11.1.2 BUILDING AND INSTALLING

Once the prerequisites have been installed, building the applications is straight-forward:

```
cd <NDI_SDK>/fpga_reference_design/src/cpp
make
make install
```

The applications will be installed with setuid privileges to the `/usr/local/bin/` directory.

23.11.2 USAGE

See the `README.md` files in each subdirectory for usage details specific to each available application.

23.12 PETALINUX PROJECTS FOR NDI® XILINX HARDWARE PLATFORMS

All steps below require you have an appropriate version of the Xilinx PetaLinux tools installed on your development machine. Refer to Xilinx UG1144 "PetaLinux tools Documentation: Reference Guide" for detailed instructions on installing PetaLinux and the required prerequisites.

23.12.1 DIRECTORY STRUCTURE:

Directory	Contents
ZCU104.2018.1	PetaLinux 2018.1 project for the Xilinx ZCU104 (Encode)
ZCU104-Dec.2018.1	PetaLinux 2018.1 project for the Xilinx ZCU104 (Decode)
Zybo-Z7-20.2018.1	PetaLinux 2018.1 project for the Digilent Zybo-Z7-20
Zybo-Z7-20-Lite.2018.1	PetaLinux 2018.1 project for the Digilent Zybo-Z7-20 with 16-bit SDRAM
ZCU104-Dec.2019.2	PetaLinux 2019.2 project for the Xilinx ZCU104 (Decode)
ZCU104-Enc.2019.2	PetaLinux 2019.2 project for the Xilinx ZCU104 (Encode)
Zybo-Z7-20.201.2	PetaLinux 2019.2 project for the Digilent Zybo-Z7-20
Zybo-Z7-20-Lite.2019.2	PetaLinux 2019.2 project for the Digilent Zybo-Z7-20 with 16-bit SDRAM

23.12.1.1 NOTES

The initial release of the NDI Advanced SDK used Petalinux 2018.1, the current release at that time. These projects have been updated to use Petalinux 2019.2. While the 2018.1 projects are still included, they should be considered deprecated and are likely to be removed in a future release.

23.12.2 BUILDING AN EXISTING PROJECT

To build one of the existing PetaLinux projects provided for the supported development boards, use the following general procedure:

```
# Setup the PetaLinux environment
source /path/to/petalinux/version/settings.sh

# Change to the appropriate project directory
cd projectdir/

# Run petalinux-configure to populate the project structure
petalinux-configure

# Build the project
petalinux-build

# Update the boot files
./boot.sh
```

23.12.2.1 NOTES

- The `petalinux-build` step will occasionally fail when performing a full build from scratch. Simply re-run the `petalinux-build` command and the build will continue. Sometimes it is necessary to re-run the `petalinux-build` step several times.
- If the `petalinux-build` step repeatedly fails in the same place, examine the log files as there may be missing dependencies. In particular, the PetaLinux 2018.1 tool requires the libgtk2.0-0 package which is not listed as a dependency in the Reference Guide.
- Output files will be in the `projectdir/images/linux/` directory.
- `projectdir` naming convention is `<platform>.<petalinux version>`

23.12.3 BUILDING A NEW PROJECT FROM SCRATCH

If you are not using one of the supported development boards you may need to build a PetaLinux project from scratch. Full details for working with PetaLinux can be found in Xilinx UG1144 "PetaLinux Tools Documentation: Reference Guide". A brief summary of one possible work flow is provided below.

- Build the hardware project
  Create a Vivado project as required for your system and build a bit file.
- Export the hardware to the SDK
  In Vivado, select the `File -> Export -> Export Hardware` menu option
- Create a new PetaLinux project

```
# For Zynq-7000
petalinux-create -t project -n <projectname> --template zynq

# For Zynq Ultrascale
petalinux-create -t project -n <projectname> --template zynqMP
Import the hardware definitions
```

```
cd <projectname>
petalinux-config --get-hw-description=/path/to/vivadoproject/project.sdk
```

- Customize the PetaLinux project (optional)

```
# Configure petalinux settings (eg: boot loader options, kernel command line, etc)
petalinux-config

# Configure the Linux kernel
petalinux-config -c kernel
```

- Customize system-user.dtsi to reflect the FPGA hardware
  Xilinx IP cores should be automatically added to the device tree, but details for any custom FPGA logic must be added manually. Details will depend on the specifics of your FPGA project. Refer to the existing projects for examples:

```
projectdir/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi
```

- Build the PetaLinux project

```
petalinux-build
```

Note: This command will occasionally fail with the error "timeout while establishing a connection with SDK". If this happens simply run the build command again.

- Create boot files
  The generated boot loader files and possibly the FPGA bit file need to be packaged into files suitable for copying to your boot media. The specific command options needed will depend on the configuration options chosen for your system. The following example is for a uSD image with the boot loader programming the FPGA prior to booting the Linux kernel:

```
petalinux-package --boot --fpga /path/to/fpga.bit --uboot

# If you have previously generated the boot files and want to rebuild them,
# you need to add the --force flag
petalinux-package --boot --fpga /path/to/fpga.bit --uboot --force
```

23.12.4 REBUILDING AFTER CHANGES

23.12.4.1XILINX FPGA IP ADDED/REMOVED

If you add or remove any Xilinx IP blocks with Linux kernel drivers, the PetaLinux project needs to be updated to incorporate the changes. The easiest way to do this is simply follow the directions for "Building a new project from scratch", above, skipping the petalinux-create command. This will complete much faster than the initial build, since most of the build artifacts are cached and can be reused.

The minimum process to update a project with new hardware definitions is:

- Export the hardware to the SDK
- In Vivado, select the `File -> Export -> Export Hardware` menu option
- Import the updated hardware definitions

```
cd <projectname>
petalinux-config --get-hw-description=/path/to/vivadoproject/project.sdk
```

23.12.4.2 FPGA BIT FILE UPDATED

If you update the FPGA bit file with hardware changes that do not alter the automatically generated device-tree, the update process is much simpler. Simply rebuild the boot files using the updated bit file:

```
# Rebuild the boot files
petalinux-package --boot --fpga /path/to/fpga.bit --uboot --force
```

If you are using the default directory structure, you can simply run one of the provided boot scripts:

```
cd <projectname>
./boot.enc.sh    # For NDI Encode
./boot.dec.sh    # For NDI Decode
```

The generated boot files will be in the `image/linux` directory, and will also be copied to a subdirectory based on the FPGA bitfile chosen (Encode/Decode & Full/Lite).

23.12.4.3 DEVICE TREE CHANGES

If you update the system-user.dtsi file, the system device tree needs to be updated and the changes need to be incorporated into the boot loader.

```
# Rebuild just the boot loader
petalinux-build -c u-boot

# Rebuild the boot files
petalinux-package --boot --fpga /path/to/fpga.bit --uboot --force
```

23.13 USD IMAGE BUILDER

Scripts to build uSD images for NDI® demo platforms based on Debian.

File	Description
README.md	This file
CHANGELOG.md	List of notable changes
composite.armhf.conf	Config file for combined Zybo-Z7 & SoCKit image
zcu104.conf	Config file for ZCU104
zybo_z7-20.conf	Config file for Zybo-Z7-20
build.sh	Main image build script
common.sh	Functions common to several scripts
boot.sh	Extracts boot files from a PetaLinux project

debootstrap.sh	Creates a Debian root filesystem from scratch
second-stage.sh	The debootstrap second-stage and rootfs customizations
mk.uSD.sh	Creates a uSD image from boot files and a rootfs
part.txt	Partion table for the uSD
update.tgz.sh	Example to generate optional tgz files

The build scripts are based on a config script which provides details needed to create an image. The main details required are a pointer to the PetaLinux project for the boot files and various architecture specific information (eg: the Debian architecture name and the qemu static binary to use for an emulated chroot environment). Various other settings may be tweaked as well, including the uSD target size, system host name, etc. Refer to the existing *.conf files for details.

23.13.1 BUILD AN IMAGE

To build an image from scratch, simply run the build.sh script with the desired configuration specified:

```
./build.sh zcu104.conf
```

23.13.2 REBUILD AN IMAGE

It is often not necessary to run all steps from scratch. Once an initial build has created the required bootfs and rootfs directories, each component of the uSD image may be updated independently. The components that makeup the uSD image are:

- bootfs.<board>/

  This directory contains the boot files and linux kernel modules extracted from the PetaLinux project. This directory is created and updated by running the `boot.sh` script.

- root.<deb_arch>.tgz

  This file is manually generated and if present will be extracted to the root directory of the target file-system by the root user. This is currently used to install pre-compiled NDI example binaries to /usr/local/bin and the required shared libraries to /usr/local/lib.

- This file is processed by the `second-stage.sh` script, so any changes require re-running `debootstrap.sh` or manually applying the changes to the rootfs directory.

  Note this file should include full paths preceded by a leading ./ eg: `./usr/local/lib/file.so`. Below is one example of how to properly create this file:

```
# Start at the root directory
cd /

# Create a tar archive with a leading ./ and put it in /tmp
tar -czvf /tmp/root.armhf.tgz ./usr/local/bin/*
```

- user.<deb_arch>.tgz

  This file is similar to the `root.<deb_arch>.tgz` file, above, except it is extracted into the default user's home directory by the default user. This file is used to install the HDMI start and stop scripts for the ZCU104, and is unused by the Zybo-Z7-20 example. Place any files that need to be owned by the default user and thus cannot be placed in the root archive (since the default username, uid/gid, and home directory may change) into this archive.

  This file is processed by the `second-stage.sh` script, so any changes require re-running `debootstrap.sh` or manually applying the changes to the rootfs directory.

- rootfs.<deb_arch>/

  This directory contains a stock Debian install generated by debootstrap along with some modifications required to make a usable image (eg: create /etc/fstab and enable networking) and tweaks required for this example (eg: build and install the libuio package). This directory is generated by running the `debootstrap.sh` script, while most of the customizations to the rootfs occur in the `second-stage.sh` script.

  Running debootstrap.sh deletes any existing rootfs and recreates it from scratch, which is a lengthy process and does not allow for modifications other than editing the `second-stage.sh` script. Once created, however, the rootfs directory may be edited manually, just be careful with file-system permissions and user ids.

  It is also possible to chroot to the rootfs and execute native commands, eg:

  ```
  # Setup shell variables for our configuration
  source zcu104.conf

  # Open a native shell in the rootfs
  sudo LANG=C.UTF-8 chroot $ROOTFS $QEMU /bin/bash
  ```

Once any desired changes have been made to the above components, a new image can be created by running:

```
sudo ./mk.uSD.sh config.conf
```

23.14 USD IMAGES FOR NDI DEMO PLATFORMS

23.14.1 LOGIN DETAILS

```
Default username: debian
Default password: tempppwd
sudo is enabled without password for root access
root login is disabled
```

23.14.2 OBTAINING THE USD IMAGE FILES

The uSD images may be downloaded via the following URLs:

Manufacturer	Board	uSD Image Download URL
Digilent	Zybo-Z7-20	http://new.tk/NDIUSDZYBO
Terasic	SoCKit	http://new.tk/NDIUSDZYBO

Xilinx	ZCU104	http://new.tk/NDIUSDZCU

NOTE: The same uSD image is used for both the SoCKit board (Intel FPGA) and the Zybo-Z7-20 board (Xilinx FPGA). The uSD image contains boot files for both systems and may be used without modification with either board.

Each uSD image is a zip archive of three files:

Extension	Description
.img.xz	The image file (xz compressed)
.bmap	A block map file for use with bmaptools (highly recommended)
.md5	Checksum of the above two files

Most users will likely only need to use the img.xz file.

23.14.3 WRITING THE IMAGE TO A USD CARD

Use of bmaptool at a Linux command line is the most efficient way to write the image file to a uSD card. If this is not an option or is considered too complex, any uSD imaging tool which supports xz compression can be used. Etcher is a good choice if you do not already have a preference.

23.14.4 BOARD DETAILS

Connections and settings required to boot using the uSD images. Refer to the vendor documentation for full details.

23.14.4.1 DIGILENT ZYBO-Z7-20

A. AVAILABLE EXAMPLES

Three FPGA examples are available for the Zybo-Z7 board:

- Decode.Xil: 4-core NDI Decoder with 1 GB of 32-bit SDRAM
- Encode.Xil: 4-core NDI Encoder with 1 GB of 32-bit SDRAM
- Enc-Lite.Xil: 2-core NDI Encoder with 512 MB of 16-bit SDRAM (1 SDRAM chip is unused)

The default uSD image uses the Enc-Full files, however files for all three examples are included on the uSD card. To change the example design, simply update the files in the `/boot` directory and reboot:

```
# Use the Decode example with 4 cores and 1 GB of 32-bit SDRAM
sudo cp /boot/Decode.Xil/* /boot/

# Use the Encode example with 4 cores and 1 GB of 32-bit SDRAM
sudo cp /boot/Encode.Xil/* /boot/

# Use the "Lite" Encode example with 2 cores and 512 MB of 16-bit SDRAM
sudo cp /boot/Enc-Lite.Xil/* /boot/
```

The "Enc-Lite" Encode example is intended to allow performance evaluation of a typical "minimal" Zynq platform that uses only a single SDRAM memory chip (16-bit memory bus). This design is still capable of processing 1080p60 video.

The "Full" Encode example will have lower latency at all resolutions and can in theory compress up to 4Kp60 4:2:0 video (with enough available SDRAM bandwidth).

A source of 4K video would also be required (eg: Analog Devices ADV7619, ITE IT68059) since the HDMI I/O on the Zybo-Z7 does not support resolutions beyond 1080p60.

B. JUMPERS AND SWITCHES

Ref	Function	Setting
JP5	Boot Mode	SD
JP6	Power	WALL

C. CONNECTIONS

Ref	Function / Label	Connect to
J9	HDMI Rx	HDMI video source (NDI Encode)
J8	HDMI Tx	HDMI video output (NDI Decode)
J7	Line In	Analog audio in
J5	Headphone	Analog audio out
J3	Ethernet	Local Ethernet network
J17	Power	Appropriate 5V power supply
J12	PROG/UART	Host system USB port

D. LEDS

LED	Function
LD0	Tally (Main)
LD1	Tally (Preview)

LD2	CPU load
LD3	uSD activity
LD4	Tally (either Main or Preview)

23.14.4.2 XILINX ZCU104

A. AVAILABLE EXAMPLES

Two FPGA examples are available for the Zybo-Z7 board:

- Dec: Decode example with 4 NDI Decode cores
- Enc: Encode example with 4 NDI Encode cores

The default uSD image uses the Enc files, however files for all examples are included on the uSD card. To change the example design, simply update the files in the `/boot` directory and reboot:

```
# Use the Decode example
sudo cp /boot/Dec/* /boot/

# Use the Encode example
sudo cp /boot/Enc/* /boot/
```

B. JUMPERS AND SWITCHES

Ref	Function	Setting
J85	POR_OVERRIDE	2-3 (default)
J12	SYSMON I2C addr	1-2 (default)
J13	SYSMON I2C addr	1-2 (default)
J20	PS_POR_B	1-2 (default)
J21	PS_SRST_B	1-2 (default)
J22	Reset Seq	open (default)
SW6 1	PS_MODE	On
SW6 2	PS_MODE	Off
SW6 3	PS_MODE	Off

SW6 4	PS_MODE	Off

C. CONNECTIONS

Ref	Function / Label	Connect to
P7	HDMI Rx (bottom)	HDMI video source
J52	Power	Power supply
P12	Ethernet	Ethernet network
J164	JTAG/UART	Host system USB port

D. LEDS

LED	Function
LED0	Tally (Main or Preview)
LED1	Tally (Main)
LED2	Tally (Preview)
LED3	CPU load

23.14.4.3 TERASIC SOCKIT

The NDI Decode demo uses the on-board VGA output, but also drives an optional DVI-HSMC daughter card which is capable of supporting higher resolutions. The DVI-HSMC daughter card will be required for the upcoming NDI Encode examples, as the SoCKit has no on-board video input.

A. AVAILABLE EXAMPLES

Currently, only the NDI Decode example is supported on the SoCKit board. An Encode example will be provided in a later release.

B. JUMPERS AND SWITCHES

Ref	Function	Setting
SW4.1	JTAG_HPS_EN	Off (default)

SW4.2	JTAG_HSMC_EN	On (default)
SW6.1	MSEL0	Off
SW6.2	MSEL1	On
SW6.3	MSEL2	On
SW6.4	MSEL3	On
SW6.5	MSEL4	On
SW6.6	N/A	On
J17	BOOTSEL0	1-2 (default)
J19	BOOTSEL1	2-3 (default)
J18	BOOTSEL2	1-2 (default)
J15	CLKSEL0	2-3 (default)
J16	CLKSEL1	2-3 (default)
JP2	HSMC VCCIO	5-6 (default)

C. CONNECTIONS

Ref	Function / Label	Connect to
J6	Line In (Blue)	Analog audio in
J7	Line Out (Green)	Analog audio out
J10	VGA Connector	Analog video out
J14	HSMC Connector	HSMC-DVI (optional)

23.14.5 USING THE IMAGE

Insert the programmed uSD card into the development board and boot as usual. It is recommended you connect to the serial terminal (via USB). All images use the following console settings:

115200 baud, 8 data bits, 1 stop bit, no parity

The system will boot and display the IP address, as well as the default username and password at the serial console login prompt. You may login either via the serial console or remotely via ssh.

23.14.5.1 FIRST BOOT

The initial boot will take somewhat longer than normal as a new set of ssh keys are generated and the root partition is expanded to fill the uSD card.

23.14.6 RUNNING THE EXAMPLE NDI ENCODE APPLICATION

An example NDI source application is provided. This application is launched automatically at boot and may also be run from the command line. The utility is named ndi_encode and is capable of using a generated video pattern or live HDMI input as the video source.

The application includes various command-line options to control the initial video mode as well as a command interpreter that can be used to change the operating mode at run time. To launch the demo from the command line, simply run one of the following commands:

```
# Live HDMI input:
ndi_encode

# Pattern generator:
ndi_encode -P
```

Once running, the operating mode can be changed by typing commands at the console. Type "help" to get a list of commands. If running with HDMI input on the ZCU104, the second virtual serial port is used as a console for the Cortex-R5 running the HDMI monitoring software. Connection details are the same as the Linux console:

115200 baud, 8 data bits, 1 stop bit, no parity

The R5 will report changes in the incoming HDMI stream on this port.

23.14.6.1 KNOWN ISSUES AND LIMITATIONS

While the NDI Encoder core is fully functional the example software and demo platforms currently have some limitations:

- This version of the NDI Advanced SDK is designed for development use and will run on a stream for 30 minutes. For a commercial use license, please email sdk@ndi.tv.
- The HDMI input hardware for the ZCU104 uses an evaluation license, so HDMI video will stop being received and the image will turn "green" after apx. 1 hour of operation. Reprogram the FPGA (eg: reboot) to restore normal operation.
- While HDMI input on the ZCU104 is functional, the HDMI management software (running on one of the R5 cores) does not communicate with the ndi_encode application. The video format tracking hardware is used to detect video format changes, however more details regarding the HDMI signal are available via the HDMI management software.

23.14.7 RUNNING THE EXAMPLE NDI DECODE APPLICATION

An example NDI receive application is provided. This application is launched manually by executing `ndi_decode`. When launched with no options the utility will attempt to locate NDI sources on the local network and will automatically connect to the first source found. A specific NDI source may be specified using the -s parameter:

```
ndi_decode -s "NDI Device (Chan 1)"
```

NOTE: The default boot files on the uSD image must be switched to the NDI Decode version prior to running the ndi_decode application. See "Available Examples", above for details.

23.14.7.1 KNOWN ISSUES AND LIMITATIONS

- Streams with alpha are decoded properly, but alpha data is not currently written to memory
- "Scaling" is performed by manipulating line stride when the NDI stream and video output resolutions do not match.
- Format conversion is not performed between 4:2:2 and 4:2:0 video sources. If the output video format does not match the NDI stream format, a warning is displayed on the Linux console.

APPENDIX A – CHANGES

A.1. NDI® ADVANCED SDK

Versions: Major.Minor.Patch

Changelogs are synchronized for Major.Minor version numbers, but each project has it's own patch version (eg: src/fpga might be version 1.1.2, and src/cpp could be 1.1.4).

A.1.1. [1.4.0] - 2020-03-25

A.1.1.1 ADDED

- NDI Decode support

A.1.1.2 CHANGED

- Updates for NDI v4.5
- Xilinx & Petalinux projects updated to 2019.2

A.1.2. [1.3.0] - 2019-07-08

A.1.2.1 CHANGED

- Updates for NDI v4.0
- Refactoring to support both encode and decode

A.1.2.2 CHANGED

- Updates to device-tree layout and uio device names
- Zybo uSD image modified to include all three examples (Encode, Encode-Lite, Decode)

A.1.3. [1.2.1] - 2018-11-17

A.1.3.1 ADDED

- Zybo-Z7-20-Lite example design (16-bit SDRAM interface with 2 encoder cores)

A.1.4. [1.2.0] - 2018-10-03

A.1.4.1 ADDED

- Auto-detect video format
- Audio is now working

A.1.5. [1.1.1] - 2018-09-25

A.1.5.1 CHANGED

- Sub-project zip files now provided in expanded form
- Restructured project directories

A.1.6. [1.1.0] - 2018-08-31

A.1.6.1 ADDED

- Support new hardware version register

A.1.6.2 CHANGED

- Updated NDI_Demo hardware project files
- Updated PetaLinux files

A.1.7. [1.0.0] - 2018-07-13

- Initial version

A.1.8. CHANGELOG HINTS & DETAILS

- Changelog format
- Semantic Versioning