NDI™ SDK

5/16/2016

OVERVIEW

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

In the previous 'quantum shift', cameras and video devices moved from analog to digital formats. In the next phase, it's clear that their signals are destined to be carried via IP. Already today, almost every camera and video device internally utilizes computer-based systems capable of communicating to IP in some manner. Most video rendering, graphics systems and switchers run on computers too; and many are *already* interconnected over IP. Handling video over the same network opens up a world of new creative and pipeline possibilities.

However – the NDI vision is much broader and more exciting than simply substituting one means of transmission between devices for another. Consider a comparison: the Internet, too, *could* be narrowly described as a transport medium, moving data from point A to point B. Yet, by connecting everyone and everything everywhere together, the 'Net' is much, much more than the sum of its parts. Likewise, introducing video into the IP realm, with its endless potential connections, offers exponential creative possibilities along with workflow benefits.

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

NDI can operate bi-directionally over a local area network, with many video streams on a shared connection. Its encoding algorithm is resolution and frame-rate independent, supporting 4K (and beyond) along with 16 channels (and more) of floating-point audio. The protocol includes tools that implement video access rights, grouping, bi-directional metadata and IP commands. Its superb performance over standard GigE networks makes it possible to transition facilities to an incredibly versatile IP video production pipeline without negating existing investments in SDI cameras and infrastructure, or costly new high-speed network infrastructures.

LICENSE

The SDK is used by you in accordance with the license you accepted by clicking "I accept" during installation. This license is available for review from the root of the SDK folder.

For commercial distribution, it is important to ensure that you implement this within your applications in the following way. If you have any questions, comments or requests, please do not hesitate to let us know. Our goal is to provide this technology and encourage its use while at the same time ensuring that there is a consistent high quality experience for both end-users and developers.

1. **Your application must provide a link to http://NDI.NewTek.com/ in a location that is close to all locations where NDI is used/selected within the product and the documentation.** This will be a landing page that provides all information about NDI and access to the available tools we provide and any updates and news. This page will not promote any of our own commercial products (or those of others) that are not directly related to NDI functionality and support.
2. Please let us know that you have a commercial application (or interesting non-commercial one) at ndi@newtek.com. We are interested in how our technology is being used and would like to ensure that we have a full list of applications that make use of NDI technology.
3. Your application should list that it uses NDI™ by name in the product description and within the application (i.e. you should list it as an "NDI input" and not a generic "Network input"). We would like it if you place NDI onto your application launch splash screen, however this not required. Logos for NDI are provided within the SDK.
4. It is recommended, but not required that you use the NDI redistributable installer instead of including the DLLs directly. If you wish to launch this silently that will skip the EULA, you may use the /SILENT flag on our installer.
   If you wish to include the dlls as part of your own application then please be aware that they require the Visual Studio 2013 C run-times https://www.microsoft.com/en-us/download/details.aspx?id=40784.
5. Your own EULA convers the specific requirements of the NDI SDK EULA.

LIBRARIES

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

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

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

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

Because NDI will include functionality to record video files to disk we are providing the first version of the API that can read these files. This API is useful because it provides a high performance, dependency less (neither QuickTime

nor any additional DLLs are required) method of reading these files that supports files that continue to be recorded while they are being read. In general, file reading and writing is done using unbuffered I/O, pre-emptive pre-fetching, and using 4K native block sizes when available. Files are read and written using the same high performance (up to 250fps of 1080p) codec as NDI uses. The native part of each file is a MOV file, and QuickTime (and AVI) native codecs are provided using the NDI libraries pre-requisites installer; the file reader SDK does not however need these to be installed. In addition to the MOV file, there are a number of index files that are present in a recorded file to allow real-time efficient indexing; these are written either into NTFS data streams when possible or into side-car files. There is an additional beta SDK that is provided called "File Reader SDK", we would be interested in any feedback.

STARTUP AND SHUTDOWN

There are commands *NDIlib_initialize()* and *NDIlib_destroy()* which can be called to initialize and de-initialize the library. It is recommended, but not ever required that you call these. Internally all objects are reference counted and on the first object creation the libraries are initialized and on the last they are destroyed which means that these calls are called implicitly. The only side effect of this behavior is that if you have a single object that you repeatedly create and destroy that more work is done each time than is required, and these calls allow that to be avoided. There is no scenario under which these calls can cause a problem even if you were to call *NDIlib_destroy()* while you still had active objects. *NDIlib_initialize()* will return false on an unsupported CPU.

CPU REQUIREMENTS

NDI Lib is heavily optimized (much written in assembly) and while it detects architecture and uses the best path it can, the minimum required SIMD level is SSE3 (https://en.wikipedia.org/wiki/SSE3). This is present in almost all CPUs that have released that also support 64-bit instructions. NDI takes advantage of newer instructions sets that are available and will benefit particularly from SSSE3 that was introduced by Intel in 2005.

NDI-SEND

Like all of the NDI libraries, a call to *NDIlib_send_create* will create an instance of the sender, which will return an instance of type *NDIlib_send_instance_t* (or NULL if it fails) representing the sending instance. The set of creation parameters that are applied to the sender are specified by filling out a structure called *NDIlib_send_create_t*. The parameters supported are as follows:

p_ndi_name (const CHAR*)

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

p_groups (const CHAR*)

This parameter represents the groups that this NDI sender should place itself into. Groups are sets of NDI sources. Any source can be part of any number of groups, and groups are comma separated. For instance "cameras,studio 1,10am show" would place a source in three groups. On the finding side you can specify

which groups to look for, and look in multiple groups. If you specify NULL as the groups then the default groups will be used.

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

1. For “receiving”, the registry path “HKLM\SOFTWARE\NDI\Groups” for a value “Receive” which can provide a full group string that will be used by default. By setting this value all receiving on your system that has no specific group applied will occur within this defined set of groups.

2. For “sending”, the registry path “HKLM\SOFTWARE\NDI\Groups” for a value “Send” can provide a full group string that will be used by default. Much as for receiving, when this string is set and the sources on your system do not specify their own group-set, this will assign all sends on the machine to this defined set of groups.

3. If the registry key is not found, everything is assumed to be within a group called “public”.

clock_video, clock_audio (BOOL)
These specify whether audio and video "clock" themselves. When they are clocked then, by adding video frames, they will be rate limited to match the current frame-rate that you are submitting at. The same is true for audio. In general if you are submitting video and audio off a single thread then you should only clock one of them (video is probably the better of the two to clock off). If you are submitting audio and video of separate threads then having both clocked can be useful.
A simplified view on how this works is that when you submit a frame it will keep track of the time that the next frame would be required at, if you then submit a frame before this time then the call will wait until that time occurs. This ensures that if you sit in a tight loop and render frames as fast as you can go, that they will be clocked at the frame-rate that you desire. Note that combining clocked video and audio submission combined with asynchronous frame submission (see below) allows you to write very simply loops to render and submit NDI frames.

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

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

NDIlib_send_instance_t pSend = NDIlib_send_create(&create_params_Send);
if (!pSend) printf(“Error creating NDI Sender”);
```

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

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

The specific structures used to describe the different frame types are described under the section “Frame types” below. An important note of understanding is that video frames are “buffered” on an input so that if you provide a

video frame to the SDK when there are no current connections to it, when a new incoming connection is received the last video frame will automatically be sent to it, this is done without any need to recompress a frame (it is buffered in memory, compressed). The following represents an example of how one might send a single 1080i59.94 white frame over an NDI sending connection.

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

// Now send it!
NDIlib_video_frame_t video_frame;
video_frame.xres = 1920;
video_frame.yres = 1080;
video_frame.FourCC = video_frameNDIlib_FourCC_type_BGRA;
video_frame.frame_rate_N = 30000;
video_frame.frame_rate_D = 1001;
video_frame.picture_aspect_ratio = 16.0f/9.0f;
video_frame.is_progressive = FALSE;
video_frame.timecode = 0LL;
video_frame.p_data = p_frame;
video_frame.line_stride_in_bytes = 1920*4;

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

// Free video memory
free(p_frame);
```

In a similar fashion, audio can be submitted for NDI audio sending, the following will submit 1920 quad-channel silent audio samples at 48kHz;

```
// Allocate an audio frame (you would do something smarter than this !)
FLOAT* p_frame = (FLOAT*)malloc(sizeof(float)*1920*4);
::memset(p_frame, 0, sizeof(float)*1920*4);

// Describe the buffer
NDIlib_audio_frame_t audio_frame;
audio_frame.sample_rate = 48000;
audio_frame.no_channels = 4;
audio_frame.no_samples = 1920;
audio_frame.timecode = 0LL;
audio_frame.p_data = p_frame;
channel_stride_in_bytes = sizeof(FLOAT)*1920;

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

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

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

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

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

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

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

An important category of meta-data that you will receive automatically when new connections to you are established is connection meta-data as specified in the NDI-Recv section of this documentation. This allows an NDI receiver to provide up-stream details to a sender that might indicate hints as to what capabilities that the receiver might have. Examples might be what resolution and frame-rate is preferred by the receiver, what it's product name is, etc… It is important that a sender is aware that it might be sending video data to more than one receiver at a time, and in this case it will receive connection meta-data from each one of them.

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

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

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

Connection metadata is data that you can "register" with a sender and it will automatically be send each time a new connection with the sender is established. The sender will internally maintain a keep a copy of any connection metadata messages that you have and send them automatically. This is useful to allow a sender to provide downstream information at the time of connection to any device that might want to connect to it; for instance letting it know what the product name or preferred video format might be. Neither senders nor receivers are required to provide this functionality and may freely ignore any connection data strings. Standard connection meta data strings are defined in a later section of this document. In order to add a meta-data element, one can call *NDIlib_send_add_connection_metadata*, and to clear all of the registered elements, one can call *NDIlib_send_clear_connection_metadata*. An example that registers the name and details of your sender so that other sources that connect to you get information about what you are is provided below.

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

const NDIlib_metadata_frame_t NDI_connection_type = {
        // The length
        ::strlen(p_connection_string),
        // The timecode
        0LL,
        // The string
        p_connection_string
};

NDIlib_send_add_connection_metadata(pNDI_send, &NDI_connection_type);
```

ASYNCHRONOUS SENDING

It is possible to send video frames asynchronously using NDI, using the call *NDIlib_send_send_video_async*. This function will return immediately and will asynchronously 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 when compared to creating your own thread and submitting frames asynchronously with rendering. The memory that you passed to the API through the will continue to be *NDIlib_video_frame_t* pointer will continue to be used until a synchronizing API call is made. Synchronizing calls are any of:

- Another call to *NDIlib_send_send_video_async.*
  Of particular note is that you call call *NDIlib_send_send_video_async(pNDI_send, NULL)* which will wait for any asynchronously scheduled frames to completed and then return immediately. Obviously you can also submit the next frame and then it will wait for the previous frame to have been finished with and then asynchronously submit the current one. Using this in conjunction with a clocked video output results in a very efficient rendering loop where you do not need to use separate threads for timing or for frame submission. In other words, the following is an efficient real-time processing system as long as rendering can always keep up with real-time.

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

- Another call to *NDIlib_send_send_video.*
- A call to *NDIlib_send_destroy.*

TIMECODE SYNTHESIS

When sending video, audio or metadata frames it is possible to specify your own timecode for all data send. You may also specify a value of *NDIlib_send_timecode_synthesize* (defined as INT64_MAX) that will instruct the SDK to generate the timecode for you. / When you specify this as a timecode, the timecode will be synthesized for you.

If you never specify a timecode at all and instead ask for each to be synthesized, then this will use the current system time as the starting timecode and then generate synthetic ones, keeping your streams exactly in sync as long as the frames you are sending do not deviate from the system time in any meaningful way. In practice this means that if you never specify timecodes that they will always be generated for you correctly. Timecodes coming from different senders on the same machine will always be in sync with each other when working in this way. If you have NTP installed on your local network, then streams can be synchronized between multiple machines with very high precision.

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

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

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

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

NDI-FIND

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

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 of performing discovery on a local network. On mDNS initialization (often done using the NDI-find SDK), a few seconds might elapse before all sources on the network are located.

Some network routers might block mDNS traffic between network segments.

Creating the find instance works very similarly to the other APIs. One fills out a *NDIlib_find_create_t* structure to describe the device that is needed. The member functions are as follows:

show_local_sources (BOOL)

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

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

Once you have a handle to the NDI find instance, you can call *NDIlib_find_get_sources* with a timeout and it will return either when the list of sources has changed or the timeout has elapsed. The list of source is always returned. In order to immediately return with the current list of located sources specify the timeout as zero. The pointer returned by *NDIlib_find_get_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_sources*, or until the *NDIlib_find_destroy* function is destroyed.

The following code will create an NDI-Find instance, and then list the current available 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(&find_create);
if (!pFind) /* Error */;

while(true) // You would not loop forever of course !
{   // Wait up to 10 seconds for new sources.
    DWORD no_sources = 0;
    const NDIlib_source_t* p_sources = NDIlib_find_get_sources(pFind, &no_sources,
10000);

    // Display all sources
    for(DWORD i=0; i<no_sources; i++)
        printf("%s\n", p_sources[idx].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 the first return to *NDILib_find_get_sources* might not include all of the sources on the network and they will be added (or removed) as any new sources are discovered. It is common that it takes a few seconds to discover all sources on a network.

For applications that wish to list the current sources in a user interface menu, the recommended approach would be to create an *NDIlib_find_instance_t* instance when you user interface is opened and then each time you wish to display the current list of available sources you can call *NDIlib_find_get_sources* with a zero time-out.

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; if the source becomes available again it will automatically reconnect.

As with the other APIs, the starting point is to use the *NDIlib_recv_create2* function. This takes settings defined by *NDIlib_recv_create2_t,* as follows below:

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

prefer_UYVY
If this parameter is true, then you will be passed UYVY video buffers whenever the source has no alpha channel. When the source has an alpha channel, buffers will be provided in RGB+A format. UYVY video generally has higher performance; however if your application processes everything in RGB, setting this value to FALSE will ease development time significantly. Internal color conversions are highly optimized, so it is generally better to let the library pass you RGB data rather than receiving UYVY and providing your own color conversions.

bandwidth
This allows you to specify whether this connection is in high or low bandwidth mode. It is an enumeration because it is possible that other alternatives will be available in the future. For most uses you should specify *NDIlib_recv_bandwidth_highest* which will result in the same stream that is being sent from the up-stream source to you. You may specify *NDIlib_recv_bandwidth_lowest* which will provide you with a medium quality stream that takes almost no bandwidth, this is normally of about 640 pixels in size on it is longest side and is a progressive video stream, no additional resources are used on the sending side to provide preview streams.

allow_video_fields
If your application does not like receiving fielded video data then you can specify FALSE to this value and all video you receive will be de-interlaced before it is passed to you. The default value should be considered TRUE for most applications..

Once you have filled out this structure, calling *NDIlib_recv_create2* will create an instance for you. A full example is provided with the SDK: it illustrates finding a network source, and creating a receiver to view it (we will not reproduce that code here).

Once you have a receiving instance, you can query it for video, audio or meta-data frames by calling *NDIlib_recv_capture*. This function takes a pointer to the header for audio (*NDIlib_audio_frame_t*), video (*NDIlib_video_frame_t*) and metadata (*NDIlib_metadata_frame_t*), any of which can be NULL. This function can safely be called across many threads at the same time, allowing you to easily have one thread receiving video while another receives audio. The function takes a timeout value specified in milliseconds. If 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_t video_frame;
NDIlib_audio_frame_t audio_frame;
switch(NDIlib_recv_capture(pRecv, &video_frame, &audio_frame, NULL, 1000 ))
{   // We received video.
    case NDIlib_frame_type_video:
         // Process video here
         // Free the video.
         NDIlib_recv_free_video(pRecv, &video_frame);
         break;

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

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

You are able, if you wish, to take the received video, audio or metadata frames and free them on another thread to ensure that there is no chance of dropping frames while receiving them. A short queue is maintained on the receiver to allow you to process incoming data in the fashion that is the most convenient within your application. (If you always process buffers faster than real-time then this queue will always be empty and you will be running at the lowest possible latency.)

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

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

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

Connection meta-data is an important concept that allows you to “register” certain meta-data messages so that each time a new connection is established that the up-stream source (normally an NDI Send user) would receive those strings. Note that connections might be lost and established at run-time for many reasons, for instance if an NDI-Sender went offline then the connection is lost, if it comes back online at a later time then the connection would be re-established and the connection meta-data would be resent. There are some standard connection strings specified for connection metadata that are outlined in the next section. Connection meta-data strings are added with *NDIlib_recv_add_connection_metadata* that takes an *NDIlib_metadata_frame_t* structure. To clear all connection metadata strings allowing them to be replaced, call *NDIlib_recv_clear_connection_metadata*. An example that illustrates how you can provide your product name to anyone who ever connects to you is provided below.

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

const NDIlib_metadata_frame_t NDI_connection_type = {
        // The length
        ::strlen(p_connection_string),
        // The timecode
        0LL,
        // The string
        p_connection_string
};

NDIlib_recv_add_connection_metadata(pNDI_recv, &NDI_connection_type);
```

RECEIVERS AND TALLY MESSAGES

Obviously any video receiver can specify whether the source it is currently on program row or preview row. This is communicated up-stream to the sender of that source who then will indicate whether it's visible state (see the section on the sender SDK within this document.) What the sender then does it take its current tally state and echo it back to all receivers as a meta-data message of the form :

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

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

CONNECTION METADATA

Connection meta-data is data that is stored and sent automatically each time that a connection is established. This allows a sender or receiver to automatically provide a detail of its capabilities so that someone connecting to it might have some idea about what it is connected too. The following are standard meta-data connection messages that are established as part of the SDK. These are not required for anyone to receiving or sending audio or video, you are also free to define your own meta-data flags although they should not start with "ndi_", instead they should follow your company name to avoid potential clashes between tags.

VIDEO AND AUDIO FORMAT

A sender or receive can provide a recommendation of what video format it prefers, for instance a video mixing device might provide a the current format that it is running at. More than one format might be specified, they should be listed in order of preference. This might allow an up stream graphics system to automatically provide video in the correct format.

The tag names match the variable names and definitions of the *NDIlib_video_frame_t* and *NDIlib_audio_frame_t* structures exactly as documented in the next section of this document.

```
<ndi_format>
   <video_format xres="1920" yres="1080" frame_rate_n="30000" frame_rate_d="1001"
                 aspect_ratio="1.77778" progressive="true"/>
   <video_format xres="1280" yres="720" frame_rate_n="30000" frame_rate_d="1001"
                 aspect_ratio="1.77778" progressive="true"/>
   <video_format xres="720" yres="480" frame_rate_n="30000" frame_rate_d="1001"
                 aspect_ratio="1.77778" progressive="true"/>
   <video_format xres="720" yres="480" frame_rate_n="30000" frame_rate_d="1001"
                 aspect_ratio="1.33333" progressive="true"/>
   <audio_format no_channels="4" sample_rate="48000"/>
   <audio_format no_channels="2" sample_rate="48000"/>
</ndi_format>
```

PRODUCT NAME

A sender or receive can provide a product name and details that are automatically transferred to the other end of the connection. Please note that like all other connection meta-data strings, providing this information is optional although it is recommended where possible.

The fields provided are described below. All fields are considered optional and might not exist in a practical implementation, the order of the fields in the XML is arbitrary.

Tag name	Description
long_name	This is a longer description of your product name. For instance “Soundscape AudioDesignMax Pro 4.0”
short_name	This is a short description that might be displayed in an interface for informational purpose. For example “AudipDesignMax Pro”
manufacturer	Your company name. For example “Soundscape”
version	A version number, there is no particular form specified for this and so should generally be treated as a string by someone receiving this data. For instance “1A0001” or “1.012.0011”
serial_number	A serial number, there is no particular form specified for this and so should generally be treated as a string by someone receiving this data. This may be used to provide product pairing as needed. For instance “NA1050192847” or “00000000001A”
model_name	A string representation of the model number being used. For instance “TCXD8000” or “CG200”
session_name	This is a unique string that identifies the current session or show. This allows an up-stream or down-stream source to potentially use a configuration that matches the current environment.

An example connection meta-data string might be:

```
<ndi_product long_name="CoolLab Video NDI Pro 1.0" short_name="Video NDI Pro"
             manufacturer="CoolLab" version="1.0" serial_number="AA001" session_name="MyShow"
             model_name="PB65-Q"/>
```

FRAME TYPES

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

VIDEO FRAMES (NDILIB_VIDEO_FRAME_T)

xres, yres (DWORD)

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

FourCC (NDIlib_FourCC_type_e)

This is the pixel format for this buffer. There are currently two supported formats, as follows :

FourCC	Description
NDIlib_FourCC_type_UYVY	YUV 4:2:2 (Y sample at every pixel, U and V sampled at every second pixel horizontally on each line). A macro-pixel contains 2 pixels in 1 DWORD. Please see notes below regarding the expected YUV color space for different resolutions. Note that when using UYVY video, the color space is maintained end-to-end through the pipeline, which is consistent with how almost all video is created and displayed.
NDIlib_FourCC_type_BGRA	BGRA linear video. This data is not premultiplied.

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

Resolution	Standard
SD resolutions	BT.601
HD resolutions	Rec.709
UHD resolutions	Rec.2020
Alpha channel	Full 0-255 range when running 8 bit.

For the sake of compatibility with standard system components, the windows APIs expose 8 bit UYVY and RGBA video since these are common FourCCs used in all media applications. 10bit and 16bit support is available on request. The internal pipeline is maintained entirely at 16-bit precision or better, and remains in YCbCr 4:2:2 + Alpha end-to-end in all cases.

frame_rate_N, frame_rate_D (DWORD)

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

frame-rate = frame_rate_n/frame_rate_d

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

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

picture_aspect_ratio (FLOAT)

The aspect ratios defined within the SDK are the picture aspect ratios (as opposed to the pixel aspect ratios). Some common aspect ratios are presented in the table below:

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

is_progressive (BOOL)

This is used to determine whether this is a progressive video frame (TRUE), or an interleaved frame.

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

NTSC 486 lines
The best way to handle this format is simply to offset the image by one line (p_uyvy_data + uyvy_stride_in_bytes) and reduce the vertical resolution to 480 lines. This can all be done without modification of the data being passed in at all: simply change the data and resolution pointers.

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

timecode (LONGLONG, 64bit signed integer)

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

p_data (const BYTE*)

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 (DWORD)

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

AUDIO FRAMES (NDILIB_AUDIO_FRAME_T)

NDI Audio is entirely floating point (thus does not experience clipping) and has a dynamic range that is largely without reasonable limits. In order to define how floating point values map into real-world audio levels, a sine-wave that is 2.0 floating point units peak-to-peak (i.e. -1.0 to +1.0) is assumed to represent and audio level of +4dBU corresponding to a nominal level of 1.228V RMS.

It is important to understand that audio in most professional installations is assumed to have 20dB of headroom, which means that audio levels up to +24dBU will occur, corresponding to a since-wave from -10.0 to +10.0. If you want a simple “recipe” to handle audio in 16bit signed integer, with 20dB of headroom then you would use :

```
int_sample = max(-32768, min(32767, (int)(0.1f*float_sample)));
```

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

The audio sample structure is defined as described below:

sample_rate (DWORD)

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

no_channels (DWORD)

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 (DWORD)

This is the number of audio samples in this buffer. This can be any number and will be handled correctly by the NDI SDK. However when sending audio and video together, 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 (LONGLONG, 64bit signed integer)

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

p_data (const FLOAT*)

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

channel_stride_in_bytes (DWORD)

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

METADATA FRAMES (NDILIB_METADATA_FRAME_T)

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

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

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

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

length (DWORD)

This is the length of the timecode in UTF8 characters. It includes the NULL terminating character.

timecode (LONGLONG, 64bit signed integer)

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

p_data (const FLOAT*)

This is the XML message data

FILE READER SDK

A file reading SDK is provided that provides dependency free access to files recorded by the NDI recording tools.

The main structure contained within this SDK is the FileReader_Info structure, that has the following members.

updating (bool)

This tells you whether the file is currently being written too. This would mean that another application is still recording to this file and so it is expanding. In general you can use this to get the hint that you as you get towards the end of the file in playback that you should get information to determine the current length since it might have been extended.

num_frames (__int64)

The current number of video frames in the file.

xres, yres (int)

This is the resolution of this frame in pixels. For instance, xres=1920, yres=1080 could specify a 1080i or 1080p video frame.

frame_rate_n, frame_rate_d (int)

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

frame-rate = frame_rate_n/frame_rate_d

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

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

progressive (bool)

This is used to determine whether this is a progressive video frame (TRUE), or an interleaved frame.

aspect_ratio (float)

The aspect ratios defined within the SDK are the picture aspect ratios (as opposed to the pixel aspect ratios). Some common aspect ratios are presented in the table below:

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

has_audio (bool)

Does this file have an audio stream on it.

num_samples (__int64)

The current number of audio samples in the file. This can of course be a very large number.

num_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.

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.

The functionality of the file reader is quite simple. A file can be opened with a call to *FileReader_Create*. This function will return NULL if the file failed to open. Obviously the file can be closed with the matching call to *FileReader_Destroy*.

```
void* p_MyFile = ::FileReader_Create(L"My Amazing File.mov");

if (!p_MyFile) { /* File failed to open */ return; }

::FileReader_Destroy( p_MyFile );
```

Information about the file can be got at any time with a call to *FileReader_GetInfo*. A structure of type *FileReader_Info* is passed into the function and it will return the current file information. If a file is currently growing, at the time that the function is called it will tell you the current number of frames and audio samples. This operates cross process and even across machines (i.e. the file can be being recorded by one machine onto centralized storage and queried by another). Typically if the "updating" member is set to true, then the file is currently being written too and so should it should be considered that periodically you might want to check the current file length.

The current video data can be retrieved using the functions *FileReader_GetFrameYCbCr* and *FileReader_GetFrameBGRA*. You are free to read an arbitrary frame from the file at any time.

The current audio data can be retrieved using *FileReader_GetSamples*. You can read any set of audio samples from any position within the file. Because you might read past the end of the file, you are returned the number of audio samples that where actually read.

In order to convert from frame numbers to true code (timecode in 100ns units), you can use the functions *FileReader_TimeCode_from_FrameNumber* and *FileReader_FrameNumber_from_TimeCode*. This allows video and audio to be synchronized across different files and systems; the timecodes represent the real time that each frame was recorded at.

The file reader relies on NTFS data streams, or side-car files to provide information that exceeds what MOV files provide on their own, particularly when being written to on the fly. It is important that this data is preserved when copying and moving files when used by this file reader.

COMMON PROBLEMS

META-DATA DOES NOT SEEM TO BE WORKING

The most common cause of this is that you are sending meta-data that is not correctly formed XML. The NDI SDK will check that the meta-data is true XML before sending it; this is important so that any receivers on the other end would not potentially fail if they received mal-formed data.

VIDEO QUALITY AND PERFORMANCE

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

The PSNR of the codec exceeds 70dB on typical video content. Unique to this implementation is the property that once it enters the compressed video space that any further compression suffers no further generational loss at all. For instance, the following image shows the first generation, the second generation of decompression then re-compression and then the 1000th generation (!) of decompression and re-compression. Because the compression is "local", this means that even when mixing compressed video frames from different sources will not suffer any additional loss.

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

Original Image

Generation 1

Generation 1000

CONVENTIONS AND NOTES

STRING FORMATS

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

TIME-CODE

All messages of all types within NDI have an associated time-code value, defined as a 64 bit integer representing a time in 10ns intervals.

LATENCY

Latency of NDI is one field or better. Because the SDK implementation typically provides frame-at-a-time for reasons of compatibility with existing systems, the minimum latency is likely to be one frame. A hardware implementation can provide full end-to-end latency within 8 scan-lines.

AIRSEND COMPATABILITY

AirSend is a widely used API by NewTek that provides network frame-buffer output. The NDI SDK includes “replacement” versions of the AirSend DLLs that can simply be swapped with the ones on existing user or development systems, these DLLs are compatible with both AirSend and NDI sources. This wraps all NDI functionality in the AirSend interface while also providing AirSend support.

Because there are hundreds of thousands of systems that have applications that support NDI, some of which might not be being actively upgraded and maintained, a tool is provided that allows a user to update any AirSend applications on a computer system to support NDI.

CONTACT DETAILS

For all contact, support, suggestions and requests, please email: ndi@newtek.com.