NDI™ SDK QUICK START GUIDE

INTRODUCTION

This document is a very quick introduction to sending, receiving and enumerating NDI sources on a network. The goal is to illustrate how few lines of code it takes to achieve these goals. Obviously we ignore a lot of the advanced capabilities of the SDK in this document and we would refer you to the SDK documentation for the full details. There is huge flexibility around creating frames, using meta-data, controlling recording, PTZ cameras, video and audio formats and much more. We hope however that this allows you to write just a few lines of code and have video show up … and that gives you the taste of what is possible and tempt you into learning more. Above all else we recommend that you have fun and be creative … that is why NDI was created.

SENDING VIDEO OR AUDIO

We are going to write a very simple video sender that sends a single frame of video. We create an NDI sender instance, although this can be configured manually we can also just allow the SDK to choose all settings manually, so our next list would be:

```
NDIlib_send_instance_t my_ndi_send = NDIlib_send_create(nullptr);
If (!my_ndi_send) /* Failed to create sender */return false;
```

We are now going to send an HD frame of BGRA data that we already have rendered in memory at a pointer p_bgra_frame. We now initialize a frame descriptor of the frame that is going to be send with:

```
NDIlib_video_frame_v2_t video_frame_data;
video_frame_data.xres = xres;
video_frame_data.yres = yres;
video_frame_data.FourCC = NDIlib_FourCC_type_BGRA;
video_frame_data.p_data = p_bgra;
```

Please note that if you look at the definition of this structure you can specify the aspect ratio, fielding mode, line strides, meta-data and much more, but when those capabilities are not required they may simply be ignored.

You may now send this frame to NDI with a single call:

```
NDIlib_send_send_video(my_ndi_send, &video_frame_data);
```

You can send as many frames as you want in this way, on a frame by frame basis you can change any properties including resolution, color format, aspect ratio, etc… You do not need to continually send video frames because NDI will buffer the last one its elf in the NDI compressed format so that

it can be resent should anyone wish to connect to your NDI sender even when you are not sending any frames. It is very similar to send audio buffers with the structure `NDIlib_audio_frame_v2_t` that is sent with the function `NDIlib_send_send_audio_v2`.

One you wish to exit your application and no longer need the connection you may simply call the destroy function.

```
NDIlib_send_destroy(my_ndi_send);
```

This is all that is needed to implement a simply sender and although there are many other capabilities and features you can take advantage of, this would be an entirely correct application that works fully.

FINDING SOURCES

In the previous example we described how to send video over NDI. In order to discover the list of sources that are currently being sent onto the network you start by creating an NDI finder instance as follows:

```
NDIlib_find_instance_t my_ndi_find = NDIlib_find_create_v2(nullptr);
If (!my_ndi_find) /* Failed to create finder */return false;
```

Like other NDI initializations you may pass additional parameters into the `NDIlib_find_create_v2` function should you desire. Once you have a finder instance it will immediately start looking for network sources; it might take a few seconds for it to find all of them but at any time you can query it in order to get all the senders that it has found on the network:

```
uint32_t no_srcs; // This will contain how many senders have been found so far.
const NDIlib_source_t* p_senders = NDIlib_find_get_current_sources(my_ndi_find, &no_srcs);
```

`p_senders` is an array of length `no_srcs` that has all the sources found; for instance the third item on the list could be accessed as `p_senders[2]`[1]. The lifetime of the `p_senders` pointer is valid until the next call to `NDIlib_find_get_current_sources` or the finder is destroyed. Because sources on a network do not all get detected at once and new sources might come online and existing sources might go offline you can either repeat the call above each time you need the list, or you can make the following call that allows you to sleep for some period of time until the list of sources has changed.

```
bool source_list_has_changed = NDIlib_find_wait_for_sources(my_ndi_find, 2500/* 2.5 seconds */);
```

Once you no longer want to keep up to date with the available sources on the network you can simply destroy the finder with a call too.

```
NDIlib_find_destroy(my_ndi_find);
```

---

[1] Remember that in C/C++ that an array is indexed from 0.

This is all that is needed to locate sources and know when the list of sources found has changed. Once you have the list of sources that are available to you then you can use one of these to receive video as described in the next section.

RECEIVING VIDEO AND AUDIO

In order to receive a source, the first step is to create a receiver. This can be achieved with:

```
NDIlib_recv_instance_t my_ndi_recv = NDIlib_recv_create_v3(nullptr);
If (!my_ndi_recv) /* Failed to create finder */return false;
```

You may pass parameters into `NDIlib_recv_create_v3` that can specify what your preferred color space is, whether you want fielded video and more if you wish.

Once you have a receiver, you should connect it to a network sender. You first create a description of the source that you wish to connect too, assuming that you wish to connect to the source “MIX 1” and is running on a computer named “My Computer”, this would be achieved with the code

```
NDIlib_source_t my_source;
my_source.p_ndi_name = "My Computer (MIX 1)";
```

In practice, you would probably use the source returned by an NDI finder as described in the previous section. With the source ready, you would simply initiate the connection with

```
NDIlib_recv_connect(my_ndi_recv, &my_source);
```

Note that even if this source does not currently exist on the network that NDI will successfully initiate the process of connection and the moment that it becomes available it will then immediately connect automatically. If a source disappears and re-appears then NDI will automatically reconnect without you needing to have any code to initiate this. You may reconnect to other sources on the receiver at any time.

Once the connection is ready you must initiate a loop that will receive messages that come from the network, this is performed with a loop that receives frames until you no longer desire to get any.

```
while( your_application_wants_to_receive() )
{ NDIlib_video_frame_v2_t video_recv;
  NDIlib_audio_frame_v2_t audio_recv;
  switch(NDIlib_recv_capture_v2(my_ndi_recv, &video_recv, &audio_recv, nullptr, 1500))
  {  case NDIlib_frame_type_video:
               process_your_video(&video_recv);
               NDIlib_recv_free_video_v2(my_ndi_recv, &video_recv);
               break;
     case NDIlib_frame_type_audio:
               process_your_audio(&audio_recv);
               NDIlib_recv_free_audio_v2(my_ndi_recv, &audio_recv);
               break;
  } // switch
} // while
```

This loop will ask the SDK for a frame with a timeout of 1500 milliseconds (1.5 seconds) and if there are any video, audio or meta-data frames queued up or which are received in this time the descriptions will be placed in the relevant structure and the frame type received would be returned. The structures refer to memory allocated when the frames are received and so you must free this memory with the relevant free function. Frames are received and processed by NDI on at least one separate thread that is part of the receiver, meaning that frames are not dropped easily in this loop if your processing might not always return immediately. There is no requirement to free the frames before you capture the next one, allowing a host application to queue them or process them asynchronously on another thread[2].

Once you are complete with the receiver you simply destroy it with

```
NDIlib_recv_destroy(pNDI_recv);
```

[2] NDI receive calls are re-entrant so you are free to receive for instance audio and video on separate threads, you may free the data on any threads if you place them on a queue and process asyncronously.