Low Level Audio in PEPPER

Objective:

- First pass implementation is to provide 16bit, stereo, CD (44.1kHz) or DAT (48kHz) quality audio.
- Deliver low latency audio on systems that have good audio drivers. (<=20ms)
- Be flexible enough to scale additional features in future versions
    - additional sample formats
    - additional speaker configurations
    - audio input
- It will be the application's responsibility to provide higher level audio functionality.
- If an audio device is available, this API will guarantee
    - stereo output (regardless of physical speaker configuration)
    - the availability of both 44.1kHz and 48kHz output.
    - the availability of int16 as a sample format.
- The basic audio interface is a pure "C" API.
- Only linear PCM formats will be supported.
- Straight forward to secure.
- Implemented in Pepper's device API

The browser might be responsible for applying additional user volume or muting control, and may upsample/downsample the audio data before mixing with other browser audio sources.

All future versions of PEPPER will support version 1 (this proposal) as a bare minimum.

Trusted PEPPER modules will be able to open more than one audio context.
Untrusted PEPPER modules might be restricted to a single audio context.
In either environment, it is assumed multi-threading is available.

This document contains two models - one based on callbacks, and one based on a blocking push model. While both models are fundamentally the same, they do carry some subtle differences:

- Blocking Push API provides more control, and assumes threading will be explicitly handled by the application. When writing trusted PEPPER modules without a platform independent threading library, this may imply the application writer to #if WIN32, #if PTHREADS, etc. around the application's audio thread setup code.
- Callback API does not require application to setup/teardown audio thread, and the API entry points are all on the NPAPI thread -- only the callback occurs on an implicit, dedicated audio thread.
- There might be some subtle implementation details when it comes to how each model needs to signal between the audio device and the PEPPER audio interface. These signals will need to be as fast and low latency as possible, so minimizing how many signals are required will be important.

Background:

Native Client plug-ns have up until now used an audio API very similar to a blocking flushContext() call, usually done on a thread dedicated to audio. The API call was implemented as a syscall, and at the service runtime (kernel) level used SDL to deliver low latency audio across multiple platforms. As Native Client moves into a more secure sandbox environment, untrusted code will no longer be able to communicate directly with audio devices in the same process. In the new audio model, trusted code in another process will be attached to the audio device, and the sandboxed Native Client untrusted application will use shared memory and sockets to communicate its audio needs outside the sandbox. The new API for this model will be part of PEPPER, a variant of NPAPI that is designed to be platform neutral.

Please refer to Appendix A for a list of links pertaining to audio.
Please refer to Appendix B for PEPPER design wiki -- where this document will migrate once approved.

Both models will need additional PEPPER events. Please refer to Appendix C for audio related events.
Both models support multi-channel configurations. Please refer to Appendix E for multi-channel output scenarios.
Please refer to Appendix F for input scenarios.
Please refer to Appendix G for contributers.

Changes to Pepper's Device API

Previously, Pepper's device API consisted of the methods initializeContext, flushContext, and destroyContext. Audio extends this API to queryCapability, queryConfig, initializeContext, flushContext, getState, setState, and destroyContext.

> queryCapability - ask a device about a specific capability
> queryConfig - ask a device about a configuration (a set of capabilities)
> initializeContext - given a configuration, initialize a device context
> flushContext - flush context to device, either blocking or with a completion callback
> getStateContext - get state information from device context
> setStateContext - set device context state
> destroyContext - shutdown and destroy device context

NPAPI Thead, Plug-ins, and Plug-in Instances

The following device methods must be called from the NPAPI thread (Tp). They should not be called from any other thread, unless it is via NPN_ThreadAsyncCall.

> queryCapability, queryConfig, initializeContext, getStateContext, setStateContext, destroyContext

The only device method that can be called from another thread is flushContext when using the push model. When using the callback model, it is expected that the user supplied callback will be invoked on a thread dedicated to audio. This dedicated audio thead is not an NPAPI thread.

When creating an audio context, usually this is done as part of a plug-in instance. In untrusted Pepper modules, there will be one instance per sandboxed process. However, this could change in the future, and an untrusted Pepper plug-in could host multiple instances in the same sandbox. In this case, it may be desirable to have one audio device, with it's own application supplied mixer, to be shared amongst mutliple untrusted plug-in instances. This audio context should be created at Plug-in initialization, using NULL as the npp instance argument.

Obtaining Pepper Extensions and the Pepper Audio Interface

Regardless of callback vs. push model, first the Pepper audio interface must be acquired.

```
NPPepperExtensions *pepper;
NPAudio *audio = NULL;
/* Get PEPPER extensions */
NPERR ret = NPN_GetValue(instance, NPNVPepperExtensions, &pepper);
if (NPERR_NO_ERROR == ret) {
  /* successfully obtained Pepper extensions, now acquire audio... */
  audio = pepper->acquireDevice(NPAudioDeviceID);
} else {
  /* Pepper extensions are not available */
}
/* check for acquisition of audio device interface */
if (NULL != audio) {
  /* Pepper audio device interface is available */
  ...
```

Once an interface to the audio device is acquired, an application can query capabilities of the device, and from there, create a configuration (a set of capabilities.) In this simple example, the application picks 44.1kHz, stereo output using int16 sample types. It uses queryCapability to retrieve a recommended sample frame count for the output buffer.

```
  ...
  /* fill out request, use sample buffer size from configuration */
  NPAudioConfig request;
  NPAudioConfig config;
  int32 ret;
  request.sampleRate = NPAudioSampleRate44100Hz;
  request.sampleType = NPAudioSampleTypeInt16;
  ret = audio->queryCapability(npp,
    NPAudioQuerySampleFrameCount44100Hz, &request.sampleFrameCount);
  request.outputChannelMap = NPAudioChannelStereo;
  request.inputChannelMap = NPAudioChannelNone;
```

```
    request.flags = 0;
    request.callback = NULL;
    request.userData = NULL;
    ret = audio->queryConfig(npp, &request, &config);
    if (NPERR_NO_ERROR == ret) {
      /* config struct now contains a valid audio device config */
      ...
```

This example uses a very standard audio device configuration. When requesting less common configurations, an application should be prepared to work with the configuration returned by queryConfig(), which may end up being different than the requested configuration. For example, a device might return that it is capable of emitting audio at 96kHz. It could also return that it is capable of 5.1 channel output. However, when the application requests a configuration of 5.1 channel output at 96kHz, queryConfig may return back that in this case, the device is only capable of 48kHz playback over 5.1 channels. The application then has to decide, take the 5.1 output at 48kHz, or try another configuration, perhaps stereo at 96kHz, depending on which is more important - number of output channels or sample rate.

To make the most common cases easy, Pepper audio guarantees availability of 44.1kHz, 48kHz, int16 sample types, and stereo output. Pepper audio will automatically resample if needed, and do stereo->mono mixdown if needed.

If an application desires, it can query Pepper audio for the system's internal sample rate. Using this sample rate will avoid resampling (which may occur at either the device driver level or a software mixer.)

Mode One: Callback Model

The callback model has some restrictions -- all functions except the callback itself will occur on the NPAPI thread. The callback will occur on a high priority thread dedicated to serving audio. This thread is created implicitly during initialization, and cleaned up automatically at shutdown.

Example Usage

Once an audio extension is acquired via NPN_GetValue(), it can be queried, initialized, and shutdown. Stereo, int16 sample format are always guaranteed to be available.

```
  /* ...somewhere in application, from the NPAPI thread... */
  /* variable 'npp' is a NPP plug-in instance... */

  /* fill out request, use sample buffer size from configuration */
  NPAudioConfig request;
  NPAudioConfig config;
  NPAudioContext context;
  int32 ret;
  request.sampleRate = NPAudioSampleRate44100Hz;
  request.sampleType = NPAudioSampleTypeInt16;
  request.sampleBufferSize =
      audio->queryCapability(npp, NPAudioQuerySampleFrameCount44100Hz);
  request.outputChannelMap = NPAudioChannelStereo;
  request.inputChannelMap = NPAudioChannelNone;
  request.flags = 0;
  /* specify callback function to enable callback mode */
  request.callback = appCallback;
  request.userData = NULL;
  ret = audio->queryConfig(npp, &request, &config);
  if (NPERR_NO_ERROR == ret) {
    /* for this simple example, take the config we're given */
    ret = audio->initializeContext(npp,
                                 &config,
                                 &context);
    if (NPERR_NO_ERROR == ret) {
      /* audio context aquired, set state to start streaming */
      audio->setStateContext(npp, &context,
          NPAudioContextStateCallback, NPAudioCallbackStart);
      /* app_audio_callback will now be periodically invoked on */
      /* an implicit high priority audio thread */
    }
  }
```

Later on, when the PEPPER application needs to shutdown audio:

```
  /*...somewhere in application, from the NPAPI thread...*/
  audio->destroyContext(npp, &context);
```

An example callback for the Initialization function. This callback does very little work. Most of the time, the implicit thread which invokes this callback will be sleeping, waiting for the audio context to request another buffer of audio data. When possible, this callback will be occurring in a thread whose priority is boosted.

```
  void appCallback(const NPAudioContext *context) {
    /* assume using int16 format */
    int16 *outBuffer16 = (int16 *)context->outBuffer;
    for (int32 i = 0; i < context->sampleFrameCount; ++i) {
      /* generate noise */
      outBuffer16[0] = int16(rand());
      outBuffer16[1] = int16(rand());
      outBuffer16 += 2; // next interleaved sample, 2 chn stereo
    }
  }
```

Model Two: Blocking Push Model

The blocking push model allows the application to explicitly create the dedicated audio thread, and from this thread is can use blocking calls to flushContext() to push audio data. The flushContext() call is the only call that is permitted to be directly callable from a non-Tp (NPAPI) thread. We justify this because low latency audio needs to be done from a thread dedicated entirely to audio. This thread may need its priority explicitly boosted in order to satisfy scheduling demands. This model assumes the application programmer will setup the dedicated audio thread (and has access to a thread API such as pthreads.)

Example Usage

```
/* From the application's NPAPI thread... */

 /* setup request structure, use sample_buffer_size from configuration */
 NPAudioConfiguration request;
 NPAudioConfiguration config;
 pthread_t appAudioThreadID = 0;
 volatile appAudioDone = false;

 request.sampleRate = NPAudioSampleRate44100Hz;
 request.sampleType = NPAudioSampleTypeInt16;
 audio->queryCapability(npp,
                       NPAudioCapabilitySampleFrameCount44100Hz,
                       &request.sampleFrameCount);
 request.outputChannelMap = NPAudioChannelStereo;
 request.inputChannelMap = NPAudioChannelNone;
 request.flags = 0;
 /* specify NULL callback to enable blocking push model */
 request.callback = NULL;
 /* use userData field to transfer audio interface to audio thread */
 request.userData = audio;
 ret = audio->queryConfig(npp, &request, &config);
 if (NPERR_NO_ERROR == ret) {
   ret = audio->initializeContext(npp, &config, &context);
   if (NPERR_NO_ERROR == ret) {

     int p = pthread_create(&appAudioThreadID, NULL,
         appAudioThread, &context);
     /* wait for a while */
     sleep(100);
     /* notify audio thread we're done */
     appAudioDone = true;
     /* (technically, we should wait until pending flush completes) */
     audio->shutdownContext(npp, &context);
   }
 }


 /* application's thread dedicated to audio */
 void* appAudioThread(void *userData) {

   /* context is sent via pthread's userData function arg */
   NPAudioContext *context = (NPAudioContext *)userData;
   /* get pepper audio extensions (so we can call flushContext) */
   NPAudio *audio = (NPAudio *)context->userData;

   /* simple audio loop for this example, poll global variable */
   while (!appAudioDone) {
     int16 *outBuffer16 = (int16 *)context->outBuffer;
     for (int32 i = 0; i < obtained.sample_buffer_size; ++i) {
       /* example: generate some noise */
       outBuffer16[0] = int16(rand());
       outBuffer16[1] = int16(rand());
       outBuffer16 += 2;
     }
     /*
      * steam will output buffer to audio context, then block until
      * context is ready to consume the next output buffer.  Most
      * of the time, this audio thread will be sleeping in this
      * blocking call.
      */
     audio->flushContext(npp, context);
   }
   /* pthread exit point */
   return NULL;
 }
```

Switching to Low Power (Silence)

Callback Mode: While an application is performing audio output, it will receive callbacks at a regular interval. If an application needs to enter a state where audio output isn't required, it may wish to suppress the callback to save power and/or CPU cycles. The application can suppress the audio callback in two ways: 1) It can shutdown the audio context, and at a later time, when the application needs to resume audio output, it can re-initialize a new audio context. This approach may have long latency. 2) If setStateContext(npp, NPAudioContextStateCallback, NPAudioCallbackStop) is invoked on the NPAPI thread, callbacks will be suspended and the implicit audio thread will go into a sleep state. To resume audio playback and callbacks, the NPAPI thread should invoke setStateContext(npp, NPAudioContextStateCallback, NPAudioCallbackStop), which will resume callbacks in a low latency time frame.

Push Mode: In the blocking push model, flushContext() will block (sleep) until the audio context is ready to consume another chunk of sample data. If the audio thread does not call flushContext() within a short period of time (the duration of which depends on the sample frame count and sample frequency), the audio context will automatically emit silience. Normally, an audio thread will continuously make calls to the blocking flushContext() call to emit long continuous periods of audio output. If the application enters a state where no audio output is needed for an extended duration and it wishes to reduce CPU load, the application has one of two options. 1) Shutdown the audio context and exit the audio thread. When audio playback needs to resume, it can re-spawn the audio thread and re-initialize an audio context. This approach may have a long latency. 2) The application can sleep the audio thread during long periods of silence by waiting on a pthread cond_var. When the main thread is ready to resume audio playback, it can signal the condition variable to wake up the sleeping audio thread. This approach is expected to resume audio output in a low latency time frame.

Basic API

The basic Pepper API consists of: queryCapability, queryConfig, initializeContext, setStateContext, getStateContext, flushContext, and destroyContext.

```
/* min & max sample frame count */
static const int32 NPAudioMinSampleFrameCount = 64;
static const int32 NPAudioMaxSampleFrameCount = 32768;

/* supported sample rates */
static const int32 NPAudioSampleRate44100Hz = 44100;
static const int32 NPAudioSampleRate48000Hz = 48000;
static const int32 NPAudioSampleRate96000Hz = 96000;

/* supported sample formats */
static const int32 NPAudioSampleTypeInt16 = 0;
static const int32 NPAudioSampleTypeFloat32 = 1;

/* supported channel layouts */
static const int32 NPAudioChannelNone = 0;
static const int32 NPAudioChannelMono = 1;
static const int32 NPAudioChannelStereo = 2;
static const int32 NPAudioChannelThree  = 3;
static const int32 NPAudioChannelFour = 4;
static const int32 NPAudioChannelFive = 5;
static const int32 NPAudioChannelFiveOne = 6;
static const int32 NPAudioChannelSeven = 7;
static const int32 NPAudioChannelSevenOne = 8;

/* audio context states */
static const int32 NPAudioContextStateCallback = 0;
static const int32 NPAudioContextStateUnderrunCounter = 1;

/* audio context state values */
static const int32 NPAudioCallbackStop = 0;
static const int32 NPAudioCallbackStart = 1;
```

```
/* audio query capabilities */
static const int32 NPAudioCapabilitySampleRate = 0;
static const int32 NPAudioCapabilitySampleType = 1;
static const int32 NPAudioCapabilitySampleFrameCount = 2;
static const int32 NPAudioCapabilitySampleFrameCount44100Hz = 3;
static const int32 NPAudioCapabilitySampleFrameCount48000Hz = 4;
static const int32 NPAudioCapabilitySampleFrameCount96000Hz = 5;
static const int32 NPAudioCapabilityOutputChannelMap = 6;
static const int32 NPAudioCapabilityInputChannelMap = 7;

/* forward decls */
typedef struct NPAudioContext NPAudioContext;
typedef struct NPAudioConfig NPAudioConfig;

/* user supplied callback function */
typedef void (*NPAudioCallback)(const NPAudioContext *context);

/* Audio config structure */
struct NPAudioConfig {
  int32 sampleRate;
  int32 sampleType;
  int32 outputChannelMap;
  int32 inputChannelMap;
  int32 sampleFrameCount;
  uint32 flags;
  NPAudioCallback callback;
  void *userData;
};

/* Audio context structure */
struct NPAudioContext {
  NPP npp;
  NPAudioConfig config;
  void *outBuffer;
  void *inBuffer;
  void *private;
};
```

NPError queryCapability(NPP npp, int32 query, int32 *value)

```
     inputs:
          NPP npp
               Plug-in instance npp
          NPDeviceCapability query
               NPAudioCapabilitySampleRate
               NPAudioCapabilitySampleType
               NPAudioCapabilitySampleFrameCount
               NPAudioCapabilitySampleFrameCount44100Hz
               NPAudioCapabilitySampleFrameCount48000Hz
               NPAudioCapabilitySampleFrameCount96000Hz
               NPAudioCapabilityOutputChannelMap
               NPAudioCapabilityInputChannelMap
     outputs:
          *value
          Value based on input query.  See section "NPAudioCapability
          Details" for input & output values.
     returns:
          NPERR_NO_ERROR
          NPERR_INVALID_PARAM
```

NPError queryConfig(NPP npp, NPAudioConfig *request, NPAudioConfig *obtain)

```
  inputs:
       NPP npp
            Plug-in instance npp
       NPAudioConfig *request
            A requested configuration, which is a set of capabilities
  outputs:
       NPAudioConfig *obtain
            The set of capabilities obtained, which may or may not match
            request input.
  returns:
       NPERR_NO_ERROR
       NPERR_INVALID_PARAM
  notes:
       Okay if request & obtain pointers are the same.
```

NPError initializeContext(NPP npp, const NPAudioConfig *config,
NPAudioContext *context)

```
  inputs:
       NPP npp
            Plug-in instance npp
       NPAudioConfig *config - a structure with which to configure the
       audio device
            int32 sampleRate
                 Both NPAudioSampleRate44100Hz and
                 NPAudioSampleRate48000Hz will always be supported.
            int32 sampleType
                 Size and format of audio sample.
                 NPAudioSampleTypeInt16 will always be supported.
            int32 outputChannelMap
                 Describes output channel mapping.
                 NPAudioChannelStereo will always be supported, regardless
                 of the physical speaker configuration.
                 NPAudioChannelNone describes no output will occur and
                 out_buffer on the callback will be NULL.
            int32 inputChannelMap
                 Describes input channel layout.
                 NPAudioChannelNone describes no input will occur and
                 inBuffer on the callback will be NULL.
            int32 sampleFrameCount
                 Requested sample frame count ranging from
                 NPAudioMinSampleFrameCount..NPAudioMaxSampleFrameCount.
                 The sample frame count will determine sample buffer size
                 and overall latency.  Count value is in sample frames,
                 which are independent of the number of audio channels --
                 a single sample frame on a stereo device means one value
                 for the left channel and one value for the right channel.
            int32 flags:
                 Reserved, set to 0.
            callback
                 Pointer to application supplied callback function.
                      NULL: Audio context is initialized in blocking push
                      mode.
                      !NULL: Audio context is initialized in callback
                      mode.
            void *userData
                 pointer to user data for callback.  Can be NULL.
      outputs:
           NPAudioContext *context
                Filled in, used to identify audio context.
      returns:
           NPERR_NO_ERROR
           NPERR_INVALID_PARAM
      notes:
           Callable only from the NPAPI thread.
           An application is free to always select either 44.1kHz or 48kHz.
           In callback mode:
                Audio context is initialized in the stopped state.  Use
                setStateContext(npp, NPAudioContextStateCallback,
                NPAudioCallbackStart) to begin callbacks.  Repeated callbacks
                will occur until either setStateContext(npp,
                NPAudioContextStateCallback, NPAudioCallbackStop) or
                destroyContext(npp, context) is issued on the NPAPI thread.
           In blocking push mode:
                From a thread dedicated to audio, use flushContext(npp,
                context) to push the contents of context->outBuffer and
                context->inBuffer to and from the audio device.
```

NPError setStateContext(NPP npp, NPAudioContext *context, int32 state, int32 value)

```
      inputs:
           NPP npp
                Plug-in instance npp
           NPAudioContext *context
                audio context to apply state
           int32 state
                NPAudioContextStateCallback
                     Start & Stop playback when in callback mode.  Audio
                     device will emit silence and callbacks will be suspended.
           int32 value
      returns:
           NPERR_NO_ERROR
           NPERR_INVALID_PARAM
           NPERR_GENERIC_ERROR
      notes:
           Callable only from the NPAPI thread.
           Use setStateContext(npp, context, NPAudioContextStateCallback,
           NPAudioCallbackStart) to resume stopped playback.
           When stopping, after a successful return value, no callbacks should
           occur.  If a pending callback is taking too long, this function
           will fail with a return value of NPERR_GENERIC_ERROR.
           After initialization in callback mode, an audio context will be in
           the NPAudioCallbackStop state.
```

NPError getStateContext(NPP npp, NPAudioContext *context, int32 state, int32 *value)

```
     inputs:
          NPP npp
               Plug-in instance npp
          NPAudioContext *context
               audio context to apply state
          int32 state
               new state value
                    NPAudioContextStateCallback
                         Get current state of callback.
                    NPAudioContextStateUnderrunCounter
                         Get current detectable underrun count since
                         initialization.
     output:
          int32 *value
     returns:
          NPERR_NO_ERROR
          NPERR_INVALID_PARAM
     notes:
          Callable only from the NPAPI thread.
```

NPError flushContext(NPP npp, NPAudioContext *context)

```
     inputs:
          NPP npp
               For audio, this function is invoked from a non-NPAPI thread,
               so npp is ignored.
          NPAudioContext *context
               context->outBuffer
                    pointer to output data.  If this field is NULL, no output
                    (or silence) will occur.
               context->inBuffer
                    pointer to input data.  If this field is NULL, no input
                    will occur.
     returns:
          NPERR_NO_ERROR
          NPERR_INVALID_PARAM
          NPERR_GENERIC_ERROR
     notes:
          This is a blocking call.  Data in context->outBuffer is streamed to
          the audio device.  The call then blocks until the audio device is
          ready to receive the next buffer payload.  If the audio device does
          not receive data within a certain time frame, it will begin to emit
          silence.  For an application to emit continuous audio without gaps
          of silence, it will need to perform a minimal amount of computation
          between consecutive flushDevice() calls on a priority boosted
          dedicated audio thread.  If the audio device is initialized with
          both input channel(s) and output channel(s), audio can stream
          isochronously on the same flushContext() call.

          In the case of audio, it is highly recommend not to make this call
          from the NPAPI thread.

          If the audio device was initialized in callback mode,
          flushContext() will do nothing and return NPERR_GENERIC_ERROR

          If the audio device is in blocking push mode and
          destroyContext(npp, context) is performed in the NPAPI thread,
          pending blocking flushContext calls will return with
          NPERR_GENERIC_ERROR.
```

NPError destroyContext(NPP npp, NPAudioContext *context)

```
     inputs:
          NPP npp
               Plug-in instance npp
          NPAudioContext context
               audio context to shutdown
     returns:
          NPERR_NO_ERROR
          NPERR_INVALID_PARAM
     notes:
          callback mode:
               Waits for pending callback (if applicable) to complete (or
               times out)  Upon return from shutdown, no further callbacks
               will occur.
          blocking push mode:
               Does _not_ automatically terminate pending blocking
               flushContext() calls occuring other threads.  It is highly
               recommended that other threads suspend calls to flushContext()
               _before_ invoking destroyContext(npp, context).
          other:
               Callable only from the NPAPI thread.  Application's
               responsibility to cleanly bring down audio before Shutdown to
               avoid clicks / pops.
```

void (NPAudioCallback *)(NPAudioContext *context);

```
     inputs:
          NPAudioContext context
          Audio context that generated this callback.  Fields within the
          context
          context->config.sampleFrameCount
               Number of sample frames to write into buffers.  This will
               always be the obtained sample frame count returned at
               initialization.  An application should never write more
               than sample_frame_count sample frames.  An application
               should avoid writing fewer than sample_frame_count sample
               frames.  Sample frame count is independent of the number
               of channels.  A sample frame count of 1 on a stereo
               device means write one left channel sample, and one right
               channel sample.
          context->outBuffer
               Pointer to output sample buffer, channels are
               interleaved.
               For a stereo int16 configuration:
                    int16 *buffer16 = (int16 *)buffer;
                    buffer16[0] is the first left channel sample
                    buffer16[1] is the first right channel sample
                    buffer16[2] is the second left channel sample
                    buffer16[3] is the second right channel sample
                    Data will always be in the native endian format of
                    the platform.
          context->inBuffer
               audio input data, NULL if no input.
               in_buffer and out_buffer will have the same sample rate,
               sample type, and sample frame count.  Only the number of
               channels can differ (note that the channel format
               determines how data is interleaved.)
          context->userData
               Void pointer to user data (same pointer supplied during
               initialization)  Can be NULL.
     notes:
          Callbacks occur on thread(s) other than the NPAPI thread.  Deliver
          audio data to the buffer(s) in a timely fashion to avoid
          underruns.  Do not make NPAPI calls from the callback function.
          Avoid calling functions that might cause the OS scheduler to
          transfer execution, such as sleep().  Callbacks will not occur
          after return from Shutdown.  If the audio device is initialized
          with both input channel(s) and output channel(s), audio can stream
          isochronously on the same callback.
```

queryCapability() Details

This section goes into more detail on what queryCapability() returns based on the input enum.

```
notes:
     queryCapability() can be called before initializeContext to probe
     capabilities of the audio device.  queryCapability() should only be
     called from the NPAPI thread.  If the output parameter is NULL,
     queryCapability will return NPERR_INVALID_PARAM.


queryCapability(npp, NPAudioCapabilitySampleRate, &value)
     output value:
          Current 'native' sample rate of the underlying audio system.  If an
          application can use this rate, less upsampling/downsampling is
          likely to occur at the driver level.
               NPAudioSampleRate44100Hz
               NPAudioSampleRate48000Hz
               NPAudioSampleRate96000Hz
     notes:
          Both NPAudioSampleRate44100Hz and NPAudioSampleRate48000Hz are
          always available, and will be upsampled/downsampled as needed.  By
          querying for the native sample rate, an application can avoid
          implicit sample rate conversion.


queryCapability(npp, NPAudioCapabilitySampleType, &value)
     output value:
          Native sample format of the underlying audio system
               NPAudioSampleTypeInt16
               NPAudioSampleTypeFloat32
     notes:
          NPAudioSampleTypeInt16 will always be available.  By querying for
          the sample format, an application may be able to discover higher
          quality output formats.


queryCapability(npp, NPAudioCapabilitySampleFrameCount, &value)
     Query the audio system for the sample buffer frame count recommended by
     NPAudioQuerySampleRate.
     output value:
          Native sample buffer size of the underlying audio system ranging
          from NPAudioMinSampleFrameCount...NPAudioMaxSampleFrameCount.
          This sample buffer frame count relates to the sample rate returned
          by queryCapability(npp, NPAudioCapabilitySampleRate, &sampleRate)
     notes:
          Upon successful initialization of the audio system, an application
          will always be able to request and obtain the native sample buffer
          frame count when specifying the sample frequency returned by
          NPAudioQuerySampleRate.


queryCapability(npp, NPAudioCapabilitySampleFrameCount44100Hz, &value)
     Query the audio system for the sample buffer frame count to use for
     44.1kHz output.
     output value:
          Recommended sample buffer frame count ranging from
          NPAudioMinSampleFrameCount...NPAudioMaxSampleFrameCount.
     notes:
          Upon successful initialization of the audio system at 44.1kHz, an
          application will always be able to request and obtain this sample
          buffer frame count.


queryCapability(npp, NPAudioQuerySampleFrameCount48000Hz, &value)
     Query the audio system for the sample buffer frame count to use for
     48kHz output.
     output value:
          Recommended sample buffer frame count ranging from
          NPAudioMinSampleFrameCount...NPAudioMaxSampleFrameCount.
     notes:
          Upon successful initialization of the audio system at 48kHz, an
          application will always be able to request and obtain this buffer
          frame count.


queryCapability(npp, NPAudioCapabilitySampleFrameCount96000Hz, &value)
     Query the audio system for the sample buffer frame count to use for
     96kHz output.
     output value:
          Recommended sample buffer frame count ranging from
          NPAudioMinSampleFrameCount...NPAudioMaxSampleFrameCount.
     notes:
          Upon successful initialization of the audio system at 96kHz, an
          application will always be able to request and obtain this buffer
          frame count.


queryCapability(npp, NPAudioCapabilityOutputChannelMap, &value)
     Query the audio system for the output/speaker arrangement.
     output value:
               NPAudioChannelNone
               NPAudioChannelMono
               NPAudioChannelStereo
               NPAudioChannelThree
               NPAudioChannelFour
               NPAudioChannelFive
               NPAudioChannelFiveLFE
               NPAudioChannelSeven
               NPAudioChannelSevenLFE
     notes:
          Upon successful initialization of the audio system, an application
          will always be able to request and obtain this channel setup.
          Additionally, NPAudioChannelMono and NPAudioChannelStereo will
          always be available, regardless of physical speaker configuration:
               mono output -> mono speaker, or
               mono output -> center speaker, or
               mono output -> left & right speakers
               stereo output -> left & right speakers, or
               stereo output -> mono speaker
```

```
queryCapability(npp, NPAudioCapabilityInputChannelMap, &value)
     Query the audio system for the input/microphone arrangement.
     output value:
               NPAudioChannelNone
               NPAudioChannelMono
               NPAudioChannelStereo
               NPAudioChannelThree
               NPAudioChannelFour
               NPAudioChannelFive
               NPAudioChannelSeven
     notes:
          Upon successful initialization of the audio system, an application
          will always be able to request and obtain this channel setup.  If
          NPAudioChannelNone is returned, either the device has no input, or
          the implementation doesn't support audio input.
```

Appendix A - Links to audio related info

http://www.kaourantin.net/2008/05/adobe-is-making-some-noise-part-1.html
http://osdl.sourceforge.net/main/documentation/rendering/SDL-audio.html
http://stackoverflow.com/questions/1595190/sounds-effects-in-iphone-game
http://insanecoding.blogspot.com/2009/06/state-of-sound-in-linux-not-so-sorry.html
http://developer.apple.com/Mac/library/documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide/AudioUnitDevelopmentFundamentals/AudioUnitDevelopmentFundamentals.html#//apple_ref/doc/uid/TP40003278-CH7-SW12

Appendix B - PEPPER wiki

https://wiki.mozilla.org/Plugins:PlatformIndependentNPAPI

Appendix C - PEPPER Audio Events

Adds:

```
     NPEventType_Audio

     struct NPAudioEvent {
       int32 subtype;
       int32 value;
     };
```

where subtype can be:

`NPEventAudioDeviceChanged` - audio can continue to play on the current device, but an application may gain additional capability by shutting down, and re-initializing. An example of this would be a laptop currently emitting audio to headphones (stereo). The user unplugs the headphones, and plugs in an optical spdif that feeds into a home theater system. In this case, the application may wish to re-initialize to gain multi-channel output. Another example - a user plugs in a microphone and/or enables microphone permissions.

`NPEventAudioMute` - the browser has muted/unmuted audio on this plug-in instance. To save battery life, the application may wish to suspend streaming of audio while muted.

`value` is 0 - entering mute state.

`value` is 1 - leaving mute state.

Appendix D - Configuration Precedence

In cases where either the device or the implementation can't support all audio features, queryConfig will need to gracefully scale back. When scaling back features, there should be a precedence to what features should be supported vs what features should be dropped. In all cases, 44.1kHz, 48kHz and stereo output should be fully supported.

- When the application asks for simultaneous audio input and output, but the underlying system can only support one or the other, queryConfig will favor audio output over audio input.
- When the application asks for multi-channel output and a sample rate >48kHz (likely 96kHz), queryConfig will prefer lower sample rates (48kHz) over reducing the number of output channels.
- If the implementation doesn't suport a "dot one" low frequency channel, queryConfig will select the equivalent multi-channel format without the LFE channel. Applications should assume the left and right front channels are full range.

Appendix E - Multi-Channel Output Configurations

Interleave orders are fixed. Which configurations are supported depends on both the physical configuration and the underlying implementation. There is one exception: NPAudioChannelStereo will always be supported.

`Format NPAudioChannelMono`

```
            +-----+
      front | ch0 |
            +-----+




          +----------+
          | listener |
          +----------+
```

If the physical speaker arrangement is multi-channel, 1 channel output will pick the front center speaker if one is present. If no front center speaker is available, 1 channel output will occur on both the front left and front right speakers. For audio output, NPAudioChannelStereo will always be available to the application, regardless of physical speaker configuration.

Sample buffer format:

```
+-----+-----+-----+
| ch0 | ch0 | ch0 | ...
+-----+-----+-----+

<-----> single sample frame
```

Format NPAudioChannelStereo

```
front +-----+           +-----+ front
left  | ch0 |           | ch1 | right
(L)   +-----+           +-----+ (R)



            +----------+
            | listener |
            +----------+
```

If the physical speaker arrangement is mono, ch0 and ch1 will be combined and output on the mono speaker. For audio output, NPAudioChannelStereo will always be availabe to the application, regardless of physical speaker configuration.

Sample buffer interleave order is L, R:

```
      +------+------+------+------+
      | ch0  | ch1  | ch0  | ch1  | ...
      +------+------+------+------+
          sample         sample
      <-  frame 0 -> <-  frame 1 ->
```

Format NPAudioChannelThree

```
                  front
                  center (C)
front +-----+    +-----+     +-----+ front
left  | ch0 |    | ch2 |     | ch1 | right
(L)   +-----+    +-----+     +-----+ (R)



               +----------+
               | listener |
               +----------+
```

Sample buffer interleave order is L, R, C:

```
      +------+------+------+------+------+------+
      | ch0  | ch1  | ch2  | ch0  | ch1  | ch2  | ...
      +------+------+------+------+------+------+

      <- sample frame 0  -> <- sample frame 1 ->
```

Format NPAudioChannelFour

```
front +-----+                    +-----+ front
left  | ch0 |                    | ch1 | right
(L)   +-----+                    +-----+ (R)


                 +----------+
                 | listener |
                 +----------+


      +-----+                    +-----+
      | ch2 |                    | ch3 |
left  +-----+                    +-----+ right
back (Lb)                                back (Rb)
```

Sample buffer interleave order is L, R, Lb, Rb:

```
      +------+------+------+------+------+------+------+------+
      | ch0  | ch1  | ch2  | ch3  | ch4  | ch0  | ch1  | ch2  | ...
      +------+------+------+------+------+------+------+------+

      <---      sample frame 0       ---> <---   sample frame 1...
```

Format NPAudioChannelFive

```
                    front
                    center (C)
front +-----+      +-----+       +-----+ front
left  | ch0 |      | ch2 |       | ch1 | right
(L)   +-----+      +-----+       +-----+ (R)



                 +----------+
                 | listener |
      +-----+    +----------+    +-----+
      | ch3 |                    | ch4 |
left  +-----+                    +-----+ right
surround (Ls)                            surround (Rs)
```

Sample buffer interleave order is L, R, C, Ls, Rs:

```
      +------+------+------+------+------+------+------+------+
      | ch0  | ch1  | ch2  | ch3  | ch4  | ch0  | ch1  | ch2  | ...
      +------+------+------+------+------+------+------+------+

      <---     sample frame 0        ---> <---    sample frame 1...
```

Format NPAudioChannelFiveLFE

```
                    front
                    center (C)
front +-----+      +-----+       +-----+ front     +-----+
left  | ch0 |      | ch2 |       | ch1 | right     | ch3 | subwoofer (LFE)
(L)   +-----+      +-----+       +-----+ (R)       +-----+



                 +----------+
                 | listener |
      +-----+    +----------+    +-----+
      | ch4 |                    | ch5 |
left  +-----+                    +-----+ right
surround (Ls)                            surround (Rs)
```

Sample buffer interleave order is L, R, C, LFE, Ls, Rs:

```
      +------+------+------+------+------+------+------+------+
      | ch0  | ch1  | ch2  | ch3  | ch4  | ch5  | ch0  | ch1  | ...
      +------+------+------+------+------+------+------+------+

      <---       sample frame 0             ---> <---      sample frame 1...
```

Format NPAudioChannelSeven

```
                    front
                    center (C)
front +-----+      +-----+       +-----+ front
left  | ch0 |      | ch2 |       | ch1 | right
(L)   +-----+      +-----+       +-----+ (R)



                 +----------+
                 | listener |
     +-----+     +----------+     +-----+
     | ch5 |                      | ch6 |
left +-----+                      +-----+ right
surround                                  surround
(Ls)         +-----+      +-----+         (Rs)
        left | ch3 |      | ch4 | right
        back +-----+      +-----+ back
        (Lb)                      (Rb)
```

Sample buffer interleave order is L, R, C, Lb, Rb, Ls, Rs:

```
      +------+------+------+------+------+------+------+------+
      | ch0  | ch1  | ch2  | ch3  | ch4  | ch5  | ch6  | ch0  | ...
      +------+------+------+------+------+------+------+------+

      <---            sample frame 0              ---> <---      sample frame 1...
```

Format NPAudioChannelSevenLFE

```
                    front
                    center (C)
front +-----+      +-----+       +-----+ front      +-----+
left  | ch0 |      | ch2 |       | ch1 | right      | ch3 | subwoofer (LFE)
(L)   +-----+      +-----+       +-----+ (R)        +-----+
```

```
                 +----------+
                 | listener |
     +-----+     +----------+     +-----+
     | ch6 |                      | ch7 |
left +-----+                      +-----+ right
surround                                  surround
(Ls)         +-----+      +-----+         (Rs)
        left | ch4 |      | ch5 | right
        back +-----+      +-----+ back
        (Lb)                      (Rb)
```

Sample buffer interleave order is L, R, C, LFE, Lb, Rb, Ls, Rs:

```
      +------+------+------+------+------+------+------+------+
      | ch0  | ch1  | ch2  | ch3  | ch4  | ch5  | ch6  | ch7  | ...
      +------+------+------+------+------+------+------+------+

      <---                sample frame 0                   ---> <---  sample frame 1...
```

Appendix F - Multi-Channel Input Formats

Interleave orders are fixed.

Format NPAudioChannelMono - input is mono microphone input
Format NPAudioChannelStereo - input is stereo line in. Interleave order same as Appendix E.
Format NPAudioChannelFive - input is SPDIF. Interleave order same as Appendix E.
Format NPAudioChannelFiveOne - input is SPDIF. Interleave order same as Appendix E.
Format NPAudioChannelSeven - input is SPDIF. Interleave order same as Appendix E.
Format NPAudioChannelSevenOne - input is SPDIF. Interleave order same as Appendix E.

Appendix G - Contributers

Andrew Scherkus	scherkus@google.com
Brad Chen	bradchen@google.com
Carlos Pizano	cpu@google.com
Chris Rogers	crogers@google.com
Darin Fisher	darin@google.com
David Sehr	sehr@google.com
Dominic Mazzoni	dmazzoni@google.com
Ian Lewis	ilewis@google.com
Nicholas Fullagar	nfullagar@google.com
Stewart Miles	smiles@google.com