Decoder SDK Monitor 1.4

Developer Guide

Revision B

Revision History

Revision	Date	Description	Engineer
A	2018-07-27	Initial version	Lois Price Julia Wen Lore Eargle (editor)
B draft 2	2019-04-25	Release 1.3: added timestamps to Summary Report.	Lois Price Lore Eargle (editor)

Contents

List of Figures

List of Tables

1. Introduction

1.1. Purpose

The Nielsen_Decoder_SDK_Monitor package provides tools for C++ software developers to create a system that monitors Nielsen audio codes. In response to receiving small buffers of audio from the calling application, the Decoder SDK Monitor periodically invokes a callback to deliver the following:

- A list of Nielsen codes detected since the last report
- A list of alarms arising from error conditions encountered during the same period.

This document describes the concepts and usage of the SDK.

1.2. Out of Scope

This document does not describe the technical details of Nielsen audio codes.

1.3. Audience

This guide is intended for experienced C/C++ software developers.

Throughout this document, the term “you” refers to the C++ software developer who is incorporating the Nielsen_Decoder_SDK_Monitor into a monitor application.

1.4. Terminology

- This document uses the terms *audio-code monitor, monitor software*, and *monitor system* interchangeably.
- This document may use the term *Monitor SDK* or simply *SDK* to refer to the *Nielsen_Decoder_SDK_Monitor*.
- It uses the terms *watermarks* and *audio codes* interchangeably.
- It uses the terms *NAES 6* and *Nielsen Watermarks* interchangeably. The abbreviations *N6* and *NW* both represent *Nielsen Watermarks*.
- It uses the terms *timestamp* and *time code* interchangeably.
- It uses the term *CSID* to refer to a CBET Station ID.
- The narrow definition of the term *SID* is the NAES (Nielsen Audio Encoding System) Station ID

- Occasionally this document uses the term *SID* to refer to a generic station ID, regardless of whether the type is NAES or CBET.

2. Concept

2.1. Monitor Application Overview

It is assumed that your monitor operates within a professional environment where it has access to Nielsen-watermarked audio. Your monitor delivers buffers of PCM audio to the Nielsen_Decoder_SDK_Monitor libraries, which, in response deliver periodic reports of detected audio codes and alert your application to any detected problems.

2.2. Tasks Performed by Your Monitor

In order to interact successfully with the SDK, your application must perform these tasks:

- Identify the sample rate and sample size of the PCM audio that you deliver to the Nielsen_Decoder_SDK_Monitor. If the sample-size is 24 bits with 32-bit alignment, you also need to determine whether the single padding byte is positioned in the most-significant or least-significant byte. Create a CMonitorSdkParameters object and use its *set* methods to set the sample size, sample rate, and channel count.
- For each audio channel, create and initialize a class (derived from IMonitorSdkCallback) to handle all callbacks (report, alarm, and log). Note that all three callbacks are defined as pure virtual functions in IMonitorSdkCallback.
- For each audio channel, create and initialize a CMonitorApi object. The CMonitorApi constructor requires that you pass pointers to your CMonitorSdkParameters and your derived callback objects.
- Repeatedly deliver buffers of audio to the Nielsen_Decoder_SDK_Monitor library by calling CMonitorApi::InputAudioData(). If the audio that you are processing has more than one channel, extract single channels of PCM audio from each audio buffer, then deliver a single channel of audio to each CMonitorApi object that you create. Each buffer should hold less than a second of audio data.
- Handle IMonitorSdkCallback::ResultCallback() by parsing the JSON string argument, retrieving the SID, code type, timestamp, and date/time string of each audio code in the array, and reporting the codes in a way that meets the requirements of your monitor.
- Handle IMonitorSdkCallback::AlarmCallback() by parsing the JSON string argument, retrieving the SID and warning code of each item in the array, and reporting the alarms in a way that meets the requirements of your monitor.
- Handle IMonitorSdkCallback::LogCallback() by logging the individual status/error messages in a way that meets the requirements of your monitor.

2.3. Tasks Performed by the Decoder SDK Monitor

The SDK accepts a single channel of PCM audio sampled at one of the supported frequencies defined in section 4 “Audio Input.” The SDK down-samples the input audio to a 24-KHz sample rate, extracts audio watermarks, aggregates the data, then periodically reports the results to your monitor application by invoking the IMonitorSdk ResultCallback(), AlarmCallback(), and LogCallback() functions.

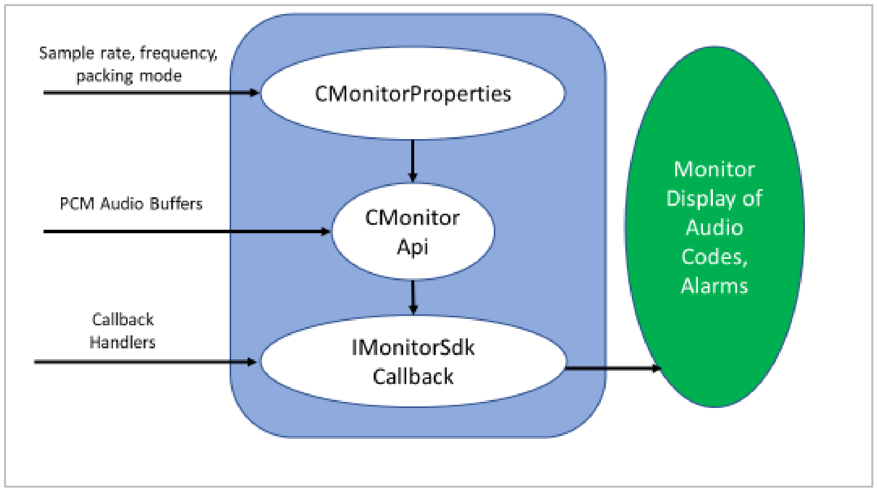

Figure 1: Decoder SDK Monitor Components

3. Nielsen Audio Codes

3.1. Audio Code Types

An audio stream may hold any of the types of Nielsen audio codes shown in Table 1:

Table 1: Audio Code Types

Audio Code Type	Description	Label
NAES 2 FD	Final distributor watermark that may exist as the only NAES 2 code in the stream, or may share the NAES 2 slot with NAES 2 PC codes.	Designated as N2FD in the JSON report string
NAES 2 PC	Program content watermark that may exist as the only NAES 2 code in the stream, or may share the NAES 2 slot with NAES 2 FD codes.	Designated as N2PC in the JSON report string

Audio Code Type	Description	Label
NAES 6 FD	Final distributor watermark that may exist as the only NAES 6 code in the stream, or may share the NAES 6 slot with up to two other NAES 6 codes (for a total of one PC and two FDs or for a total of 3 FD codes)	Designated as NWFD in the JSON string
NAES 6 PC	Program content watermark that may exist as the only NAES 6 code in the stream, or may share the NAES 6 slot with up to two other NAES 6 codes (for a total of one PC and two FD codes)	Designated as NWPC in the JSON report string
NAES 2 High Frequency	NAES 2 code used with short commercial (ad) content	Designated as N2HF in the JSON report string
NAES 6 Commercial Code	NAES 6 code used with short commercial (ad) content	Designated as NWCC in the JSON report string
CBET, Layers 2, 4, and 5		Designated as CBL2, CBL4, and CBL5
INFO SID	NAES 6 code used to uniquely identify the source encoder. If present, appears twice per hour, at 3 minutes past the hour.	Designated as Info-SID in the JSON string. The SID setting indicates that this is part 1, part 2, or part 3 of a three-part INFO SID.
RT-VOD	NAES 6 code used to indicate that the content is recently-telecast TV content, retransmitted as VOD	SID setting indicates that this is an RT-VOD flag

3.2. Station Identifiers (SIDs)

Most Nielsen audio codes include a station-identifier, a 2-byte or 4-byte value that uniquely identifies the source encoder as well as the station or content to be credited for the viewing.

For two types of audio code (RT-VOD and INFO SID), the SID field is used as an audio-code-type designator, not as a value that uniquely identifies a station or a piece of content.

Note that NAES 2 PC and NAES 6 PC may share the same SID. Likewise, the NAES 2 FD SID and the NAES 6 FD SID may have the same value. NAES SIDs usually appear in decimal format.

CBET SIDs are often referred to as *media codes or CSIDs*. They usually appear in hexadecimal format.

3.3. Timestamps

Most Nielsen audio codes include a timestamp field, a 4-byte value that, in many cases, provides information about when the content was encoded. For some types of audio codes, it is associating a date/time string with the timestamp is useful. For others, the timestamp carries information that is not directly related to a specific date and time. The paragraphs below describe the interpretation of the timestamp field for each of the audio-code types.

3.3.1. NAES 2 and NAES 6 (NW) Timestamps

In most cases, the NAES-2 and NAES-6 timestamp represents the date/time when the content was encoded *in the local time zone to which the encoder clock was set*. For example, if the monitor delivers a NAES date/time of *04/21/2019 14:33:00*, and if the NAES encoder clock was set to Pacific Daylight Time, we know that it was 2:33 PM on April 21, 2019, in the time zone to which the encoder clock was set.

NAES PC codes may be the exception to this rule. If the 4-byte timestamp field is a TIC (time in content) representing an offset from the beginning of pre-recorded content (VOD, for example), then the value should *not* be interpreted as a date/time.

3.3.2. CBET Timestamps

As opposed to NAES audio codes, CBET timestamps represent the time when the content was encoded as *Coordinated Universal Time (UTC).* For example, if the monitor SDK libraries deliver a CBET date/time of *04/21/2019 14:33:00*, we know that the content was encoded at 2:33 PM on April 21, 2019 [UTC ++00]. Because CBET date/time strings are expressed as UTC, while NAES date/time strings represent local time, reported CBET times are offset by NAES times by several hours.

When CBET audio codes are used to identify VOD (or other time-shifted viewing), the timestamp field does not translate reliably to a date/time string.

3.3.3. NW CC and N2 HF Audio Codes

For commercial codes (both NAES 6 and NAES 2), the time-code field does not represent a clock-time when the content was encoded. Therefore, no date/time string is associated with the raw timestamp.

3.3.4. NAES 6 INFO SIDs

The timestamp field in some NAES 6 audio codes (including INFO SIDs and RT-VOD codes) holds information that is not at all related to either a time-offset or a clock time. Therefore, no date/time string is associated with the raw timestamp.

4. Audio Input

4.1. Overview

If the incoming transport stream contains more than one audio stream for a single program, you must deliver only one channel of audio data to each instance of CMonitorApi.

The SDK supports uncompressed (PCM) audio streams of 24-bit or 16-bit resolution with a sample rate of 48 KHz or 24 KHz.

4.2. 16-, 24- and 32-bit Sample Size Audio Processing

The SDK processes only PCM audio of 16- and 24-bit sample sizes; however, the 24-bit audio may be packed in a 32-bit container, which requires a padding byte to be placed before or after the 24 bits of actual data. Below are the four possible layouts. All layouts assume the sample is in little endian byte order.

1. Figure 2 shows the layout for 16-bit audio packed in 2 bytes. No padding is required because the SDK does not require padding instructions.

Low-Order Byte	High-Order Byte

Figure 2: 16-Bit Audio Packed in 2 Bytes

2. Figure 3 shows the layout for 24-bit audio packed in 3 bytes. No padding is required because the SDK does not require padding instructions.

Low-Order Byte	Middle Byte	High-Order Byte

Figure 3: 24-Bit Audio Packed in 3 Bytes

3. Figure 4 shows the layout for 24-bit audio packed in 4 bytes with MSB padding. The padding byte is the most significant byte of the 32-bit sample. The order is assumed to be little-endian byte.

Low-Order Byte	Middle Byte	High-Order Byte	Padding Byte

Figure 4: 24-Bit Audio Packed in 4 Bytes with MSB Padding

4. Figure 5 shows the layout for 24-bit audio packed in 4 bytes with LSB padding. The padding byte is the least significant byte of the 32-bit sample. Order is assumed to be little-endian byte.

Padding Byte	Low-Order Byte	Middle Byte	High-Order Byte

Figure 5: 24-Bit Audio Packed in 4 Bytes with LSB Padding

4.3. Audio from Stereo Source

As each CMonitorApi object accepts only one audio channel, if the incoming audio is stereo, your application must pass only the left audio channel or the right audio channel to a single SDK object. Do not down-mix left and right channel audio into a single monaural mix for the audio input to the SDK.

Another option is to create a separate CMonitorApi object for each channel. Each instance of CMonitorApi must receive a pointer to its own unique callback object.

4.4. Audio from Multi-Channel Source

If you are receiving 5.1-channel audio, you may create a separate CMonitorApi object to process each channel, registering a unique callback object with each instance of CMonitorApi. Note, however, that the left-surround, right-surround, and low-frequency-effects channels will not have NAES 6 (Nielsen watermark) or CBET codes.

If you prefer for your software to implement only a single instance of CMonitorApi, consider combining the left and center channels. You must scale the audio prior to mixing to prevent overflow of PCM sample values. Your host application is responsible for attenuating the center channel audio -3 dB, then adding it to audio from the left channel.

Alternatively, you may use a Dolby® ProLogic® II down mix (Lt or Lo) instead of applying the above technique.

Once you have combined the left and center channels, you may deliver buffers of audio from the stream to a single CMonitorApi object.

5. SDK Package

5.1. SDK Package Description

The SDK package contains the SDK library, header files, and sample application (including source code). The SDK also holds one or more executables, each a version of the sample application created with different compiler settings. Nielsen offers at least three packages of Nielsen_Decoder_SDK_Monitor. The packages will become available in the order listed.

1. Microsoft® Windows Visual Studio® 2013
2. Microsoft® Windows Visual Studio 2015
3. Linux® CentOS

Table 2: Contents of SDK Package

Platform	Folder	Contents
All	README	Legal information
All	package\docs	Nielsen_Decoder_SDK_Monitor Developer Guide (this document). Instead of being included in the SDK zip file, this manual may be distributed separately.
All	package\apps	One or more executables compiled from the sample application, customized for the platform (Linux, Windows) that you have selected
All	\lib	Holds the libraries to which your application must link. For Windows builds, you need to link only to libMonitorSdk.lib, which includes all of the component libraries. The package includes separate libraries to support statically linked and dynamically linked C-runtime libraries. For Linux builds, link to these component libraries in addition to libMonitorSdk.a, libCBETDecoder.a, libCBETqoe.a, libNaes2HFDecoder.a, libNaes6Decoder.a, libNaes2HybridDecoder.a, and libNielsenAudioCore.a.
All	\inc	C/C++ header files required to interface with the SDK: • IMonitorSdkCallback.h – a base class from which you must derive your own callback class. It includes three pure-virtual functions that you must implement. • IMonitorSdkProcessor.h – a base class included in MonitorApi.h. Your application does not directly call any of the functions defined in this class. • MonitorApi.h – defines CMonitorApi, the primary interface to the libraries in libMonitorSdk • MonitorSdkSharedDefines.h – includes constants and type definitions shared by the SDK libraries and the sample application =================== C/C++ header files provided, with source code, to demonstrate how to use the classes defined in libMonitorSdk: • rapidjson – holds open-source header files required to parse JSON strings. Copyright / license information is included with each header file. • AudioProcessor.h – defines CAudioProcessor, the class that creates, initializes, and exercises objects whose classes are defined in libMonitorSdk. Source code is

Platform	Folder	Contents
		provided for this class. • ChannelExtractor.h – defines CChannelExtractor, a class that extracts a single channel from a buffer of multi-channel audio • MonitorApplicationDefines.h – typedefs used by the sample application • MonitorSdkCallback.h – defines CMonitorSdkCallback, a class derived from IMonitorSdkCallback to handle report callbacks, alarm callbacks, and log callbacks. • MonitorSdkOptions.h – defines CMonitorSdkOptions, a class that parses, interprets, and stores command-line arguments • MonitorSdkParameters.h – defines CMonitorSdk-Parameters, a class that stores parameters whose settings must be made available to CMonitorApi • MonitorSdkSharedDefines.h – SDK typedefs exposed to the sample application • WaveReader.h – defines CWaveReader, a class that reads header information from a WAV file, primarily to extract the channel-count, the sample size, and the sample frequency. Uses type definitions included in WavDefines.h. Like all of the classes whose source code we provide, this class is provided solely to illustrate the use of the SDK libraries/header files. It handles only very basic WAV files. For more complex WAV files, substitute your own WAV-reader class.
All	\MonitorSdkSample Application	Makefile or Visual Studio project/solution files required to build the sample application
All	\MonitorSdkSample Application	AudioProcessor.cpp – defines CAudioProcessor. See notes with AudioProcessor.h. ChannelExtractor.cpp – defines CChannelExtractor. See notes with ChannelExtractor.h. MonitorSdkCallback – defines CMonitorSdkCallback. See notes with MonitorSdkCallback.h. MonitorSdkOptions – defines CMonitorSdkOptions. See notes with MonitorSdkOptions.h. MonitorSdkSampleApplication – includes a simple main() function that exercises CAudioProcessor functionality WaveReader.cpp – defines CWaveReader. See notes with WaveReader.h

5.2. SDK Static Libraries

You may build your application on any of the following platforms:

- Microsoft Windows 8 (64-bit) or later operating system

  You may compile the sample application with Visual Studio 2013 or 2015. For each supported version of Visual Studio, there are two separate sets of libraries: one built with statically linked C-runtime, and the other built with dynamically linked C-runtime.

- Linux 64-bits (Nielsen built the package on a CentOS system)

  Note The libraries for Linux are not included in the prototype.

  Nielsen compiles the sample application with gcc/g++ (version 5.4.0). The Linux libraries are compiled with the -fPIC option, allowing you to link to executable files, static libraries, or shared libraries.

5.3. Using the SDK

Below are the objects that the sample application uses to demonstrate the functionality of the SDK. Note that the source code is provided solely as an example. You should customize the software to meet the needs of your application. You are responsible for adding the error-handling capabilities.

- IMonitorSdkCallback**:** SDK base class from which you must derive your own callback class
- CMonitorSdkCallback**:** class you derive from IMonitorSdkCallback to handle three items that the SDK delivers to you:
  - Log messages: you must store these in whatever logging scheme you support.
  - Audio-code reports (delivered in JSON array): you may display the SID/audio-code-type pairs to meet the requirements of your monitor.
  - Alarms/warnings (delivered in JSON array): you may issue these alarms/warnings in a way that meets the requirements of your monitor.
- CMonitorApi**:** primary class that your application uses to deliver audio data for processing
- CMonitorSdkOptions**:** class that reads and interprets command-line arguments

5.4. IMonitorSdkCallback.h Interface

IMonitorSdkCallback.h defines the abstract base for a callback class that your application must create. CMonitorApi uses this callback class to deliver audio-code reports (ResultCallback), log messages (LogCallback), and warnings (AlarmCallback) to your application.

In the sample application, CMonitorSdkCallback is a simple example of a derived callback class. As required, CMonitorSdkCallback provides implementations for the pure-virtual functions, ResultCallback(), LogCallback(), and MonitorCallback(). You must replace CMonitorCallback with a similar class – one that handles ResultCallback(), LogCallback(), and AlarmCallback() in a way that is appropriate for your application.

For example source code, refer to the MonitorSdkCallback.cpp/.h files.

5.5. CMonitorSdkParameters Object

The CMonitorSdkParameters class allows you to configure CMonitorApi with audio settings required to process incoming audio. The class exposes methods allowing you to set:

- Sample Size (16, 24, or 32 bits):
  - Set the sample size to 24 bits *only* if the 24-bit samples are 3-byte-aligned.
  - For 24-bit samples aligned in 4-byte blocks, set the sample size to 32, and set the packing mode to FourBytesMsbPadding or to FourBytesLsbPadding.
- Packing Mode:
  - Not required for 16-bit samples. Set to TwoBytesNoPadding as default.
  - Set to ThreeBytesNoPadding for 24-bit samples with 3-byte alignment.
  - Set to FourBytesMsbPadding or FourBytesLsbPadding for 24-bit samples with 32-bit alignment.
- Sample Rate: The number of samples per second per channel. Supported sample rates are 24 kHz and 48 kHz.

5.6. CMonitorApi Object

CMonitorApi is your primary interface to the Monitor SDK libraries. Its constructor accepts a pointer to the CMonitorSdkParameters object and a pointer to your implementation of the IMonitorSdkCallback class. If you use the default constructor, you may invoke the SetParameters() and the RegisterCallback() functions to pass pointers to the parameters and callback objects.

- void Initialize()

  Initialize() configures the application prior to the start of audio processing. You must pass pointers both to CMonitorSdkParameters and to your implementation of IMonitorSdkCallback before you call CMonitorApi::Initialize(). After the Initialize() method returns, you may call CMonitorApi::IsProcessorInitialized() to make certain that initialization succeeded.

- void GetVersion(char *pName, uint32_t size)

  GetVersion() sets <pName> to a string that identifies the version of Monitor SDK. The size argument (which should be greater than 12) indicates the number of bytes that you have allocated to pName.

- void InputAudioData(uint8_t *inBuffer, uint32_t nSize)

  InputAudioData() accepts PCM audio data from your application and submits the data to the audio decoder engine. It is a blocking call that does not return until the entire buffer has been processed.

  - inBuffer points to the audio data to be processed
  - nSize specifies the number of bytes of data in inBuffer.

  While executing the InputAudioData() function, the SDK library may invoke ResultCallback() to deliver a report to your application and/or it may invoke `AlarmCallback()` to alert your application to current error conditions. Note that the result and alarm callbacks are invoked at intervals of one minute or greater.

  It is also possible that InputAudioData() will invoke one or more calls to LogCallback() to deliver a status or error message to your application. You should include Nielsen log messages in your log file.

  In addition to decoding the incoming audio buffer, the InputAudioData function calculates the elapsed time in seconds by dividing the total number of audio bytes that it has received by the audio data rate. The calculated elapsed-time value determines the timing of the periodic reports.

Important If there is a period during which CMonitorApi receives no PCM audio, the Monitor SDK clocks do not advance during that period; as a result, no reports or alarms are generated during the period that has no audio. If you stop feeding audio to the SDK libraries for any reason, it is your responsibility to report the dropped audio feed.

6. SDK Sample Application

The sample application demonstrates how to use the Nielsen_Decoder_SDK_Monitor, but **it does not exercise all SDK capabilities**. The sample application assumes that the incoming PCM audio stream is stored as a WAV file. However, your application most likely will deliver raw PCM audio, not in WAV format. The simplified WAV reader is provided only for demonstration purposes.

6.1. Parameters Usage

The sample application generates periodic reports, each of which lists the valid audio codes detected during the past minute. Command-line options allow you to set these parameters:

```
-i <infile> -o <output report file> -l <log file> [-c selected channel] [-p packing mode]
```

Where:

- -i <file> is the full path and file name of the WAV file to process. The sample application accepts a WAV file as input and extracts audio information from its header, then passes buffers of PCM audio to the SDK libraries. The libraries detect Nielsen Watermarks and generate periodic watermark reports. Because your monitor most likely processes streaming data, you do not need to manage an input file.

- -o <output data file> is the full path name of the file that holds the generated reports and alarms. Because your monitor most likely displays reports and alarms on the monitor, you do not need to generate an output data file. However, you do need to parse the JSON string, much as the SDK callback class does.
- -l <log file> is the full path name of the file that holds error and status reports.
- -p <packing mode> is the packing mode, which is required to process audio with 24-bit samples. When processing WAV files whose audio has 24-bit (or 24-bit packed as 32-bit) samples, you must specify the packing mode. For audio with 16-bit samples, the -p option is not required, but you may use -p 3.

  3 = 16-bit samples, with 2-byte alignment

  0 = 24-bit samples, with 3-byte alignment

  1 = 24-bit samples, with 4-byte alignment, padding in most significant byte

  2 = 24-bit samples, with 4-byte alignment, padding in least significant byte
- -c <selected channel> allows the user to select the left (1) or right (2) channel for processing (assuming that the input is stereo). You could offer additional options for processing individual channels of a 5.1-channel audio stream. However, as an example, a choice of 1 or 2 is adequate to illustrate the process of channel selection. Remember that surround sound channels and the LFE channel of 5.1 content do not hold Nielsen Watermarks (NAES 6) or CBET codes.
- -h displays an explanation of command-line usage.

6.2. Sample Application Components

The sample application is comprised of these main elements:

- MonitorSdkSampleApplication.cpp file
- CAudioProcessor (.cpp and .h) class
- CMonitorSdkCallback (.cpp and .h) class
- CMonitorSdkOptions (.cpp and .h) class
- CWaveReader (.cpp and .h) class
- CChannelExtractor (.cpp and .h) class

The sample application must statically link to the SDK library (or libraries, for Linux builds), which provides the underlying functionality on which the sample application relies.

The remaining portions of Section 6 describe each of the six classes listed above. The sample application provides source code for each of these components. In order to illustrate clearly the proper usage of Monitor SDK classes, we have intentionally limited the functionality of the sample application, keeping the source code simple and easy to read.

6.2.1. MonitorSdkSampleApplication File

The MonitorSdkSampleApplcation.cpp file is the main entry point to the sample application. It instantiates and initializes the CAudioProcessor object, then calls upon the audio processor to receive and process buffers of PCM audio.

This is the very simple main() function:

```
int main(int argc, char* argv[])
{
        CAudioProcessor processor;

        // Parse, validate and process all parameters
        // ProcessParameters() returns true if successful.
        processor.ProcessParameters(argc, argv);

        // Initialize processor object with command-line settings.
        // Initialize() returns true if successful.
        if (processor.Initialize())

                // Process the entire input audio file.
                processor.ProcessData();
        processor.Release();
}
```

6.2.2. CAudioProcessor (.cpp and .h) Object

As you can see from the main() function listed in section 6.2.1, CAudioProcessor exposes just a few public methods:

- ProcessParameters() creates an instance of CMonitorSdkOptions to read and interpret command-line arguments and make them available to the CMonitorSdkParameters class.
- Initialize() creates and initializes CMonitorSdkParameters, CMonitorSdkCallback, and CChannelExtractor objects, each of which is described in Section 6 of this document.
- ProcessData() repeatedly reads buffers of data from the WAV file and delivers those buffers, in sequence, to CMonitorApi::InputAudioData().
- Release() deletes allocated objects and buffers.

6.2.3. CMonitorSdkCallback (.cpp and .h) Object

The CMonitorSdkCallback object handles report callbacks, alarm callbacks, and log callbacks by implementing the IMonitorSdkCallback interface. Before creating the CMonitorApi object, your application must create an instance of *your customized* callback class. Your application must then pass a callback-object pointer to CMonitorApi.

CMonitorSdkCallback implements the three pure-virtual methods declared in IMonitorSdkCallback.

- void ResultCallback(uint32_t elapsed_time, std::string result)

  CMonitorApi calls ResultCallback() at one-minute intervals, when it has a summary report to deliver.

  Important If your application runs for less than one minute, the SDK does not invoke the ResultCallback() function at all).

  - The first argument, elapsed_time, indicates the number of seconds that have elapsed since the beginning of processing. The SDK calculates this value by dividing the number of bytes of audio that have been processed by the data rate.

    Note The elapsed-time setting is delivered primarily for debug purposes. It is likely that your monitor displays the current system-clock time.

  - The second argument, result, is a JSON string that holds the report information. The following is an example result string:

```
{"SummaryReport":[
{"SID":"9000","AudioCodeType":1,"AudioCodeTypeDescription":"N2FD","RawTime":29
14486719,"DateTimeString":"04/22/2019 08:24:31"},
{"SID":"0x1233","AudioCodeType":12,"AudioCodeTypeDescription":"CBL5","RawTime":
1555935840,"DateTimeString":"04/22/2019 12:24:00"},
{"SID":"9000","AudioCodeType":5,"AudioCodeTypeDescription":"NWFD","RawTime":2
93617470,"DateTimeString":"04/22/2019 08:24:30"}]}
```

    The JSON array in the result string is composed of SID / code-type / code-type-description / raw timestamp / date-time string elements, each of which identifies an audio code detected during the past minute. The code type setting may range from 0 through 14 (Table 3). However, codes 0, 3, 8, and 10 are currently undefined and do not appear in any summary reports. Section 3.1 includes an explanation of the code types in Table 2.

Table 3: Code Types and Descriptions

Code Type	Code	Code Type Description
1	NAES 2 FD	N2FD
2	NAES 2 PC	N2PC
4	NAES 2 High Frequency	N2HF
5	Nielsen Watermark FD	NWFD
6	Nielsen Watermark PC	NWPC
7	Nielsen Watermark Commercial Code	NWCC
9	CBET Layer 2, SID reported in hexadecimal format	CBL2
11	CBET Layer 4, SID reported in hexadecimal format	CBL4

Code Type	Code	Code Type Description
12	CBET Layer 5, SID reported in hexadecimal format	CBL5
13	RT-VOD	RT-VOD
14	INFO-SID	Info-SID

- void AlarmCallback(uint32_t elapsed_time, std::string warning_list)

  CMonitorApi calls AlarmCallback() at one-minute intervals, but only if it has an alert / warning to deliver.

  - The first argument, elapsed_time, indicates the number of seconds that have elapsed since the beginning of processing. The SDK calculates this value by dividing the number of bytes of audio that have been processed by the data rate.

    Note The elapsed-time setting is delivered primarily for debug purposes. It is likely that your monitor will display the current system-clock time.

  - The second argument, warning_list, is a JSON string that holds an array of alarms that were active during the report period that just expired. Each element of the array lists the warning code and the SID and audio-code type to which the warning applies. An example warning string with just one array entry:

    ```
    {"WarningReport":[{"SID":"9005","AudioCodeType":"NWPC","WarningString":"Insufficie
    nt Code Count"}]}
    ```

    Table 4 shows the coded warnings that may appear in the warning list.

Table 4: Coded Warnings in Warning List

Label	Meaning
Audio Code Type Error	Watermarks with the same NAES 2 SID detected in the past minute had conflicting PC/FD types, probably due to a decoding error
Duplicate Code Error	In the past minute, the number of duplicate watermarks with this SID outnumbered the number of unique watermarks. A duplicate watermark is one that has the same SID, type, and timestamp as a previously reported watermark.
Insufficient Code Count	Issued when three or more of the last five 1-minute intervals had two or fewer watermarks with the designated SID but there were at least four watermarks altogether

Label	Meaning
Timecode Error	In the past minute, the number of watermarks with this SID that had invalid timestamps outnumbered the number of watermarks with the same SID that had valid timestamps

- void LogCallback(int_t code, const char *pMessage)

  CMonitorApi calls LogCallback() whenever it has a processing error or status message to report. Unlike the ResultCallback and AlarmCallback, the LogCallback is invoked immediately if there is a processing error or change in status.

6.2.4. CMonitorSdkOptions (.cpp and .h) Object

The CMonitorSdkOptions object reads and interprets command-line options and makes their settings available to the calling class. See Section 6.1.

6.2.5. CWaveReader (.cpp and .h) Object

The CWaveReader object parses the WAV file header to retrieve the sample size, channel-count, and sample rate of the PCM audio stream, and it makes those settings available to CAudioProcessor. The audio-processor class repeatedly calls CWaveReader:ReadBlock() to read buffers of data from the WAV PCM file.

6.2.6. CChannelExtractor (.cpp and .h) Object

The CChannelExtractor object extracts a single channel of audio from a multi-channel audio buffer and delivers the extracted audio to the calling application. The sample application uses the command-line setting -c <channel> to determine which channel of a stereo pair to extract.

- If the setting is 1, the sample application extracts the left channel.
- If the setting is 2, it extracts the right channel.

Appendix: Factors that Can Affect Decoding

Nielsen Decoder SDK Monitor supports only audio frequencies of 48 kHz and 24 kHz.

Nielsen Watermarks are designed to survive audio compression when the bitrate is maintained at the recommended level (Table 5). Code recovery may be affected when the audio is clipped or near silence.

Table 5: Acceptable Audio Compression rates for Code Recovery

Audio Compression	Compression Rate
AC3 Stereo	192 Kbps or higher
AC3 5.1	384 Kbps or higher
Enhanced AC3	192 Kbps
MPEG2 audio	192 Kbps or higher