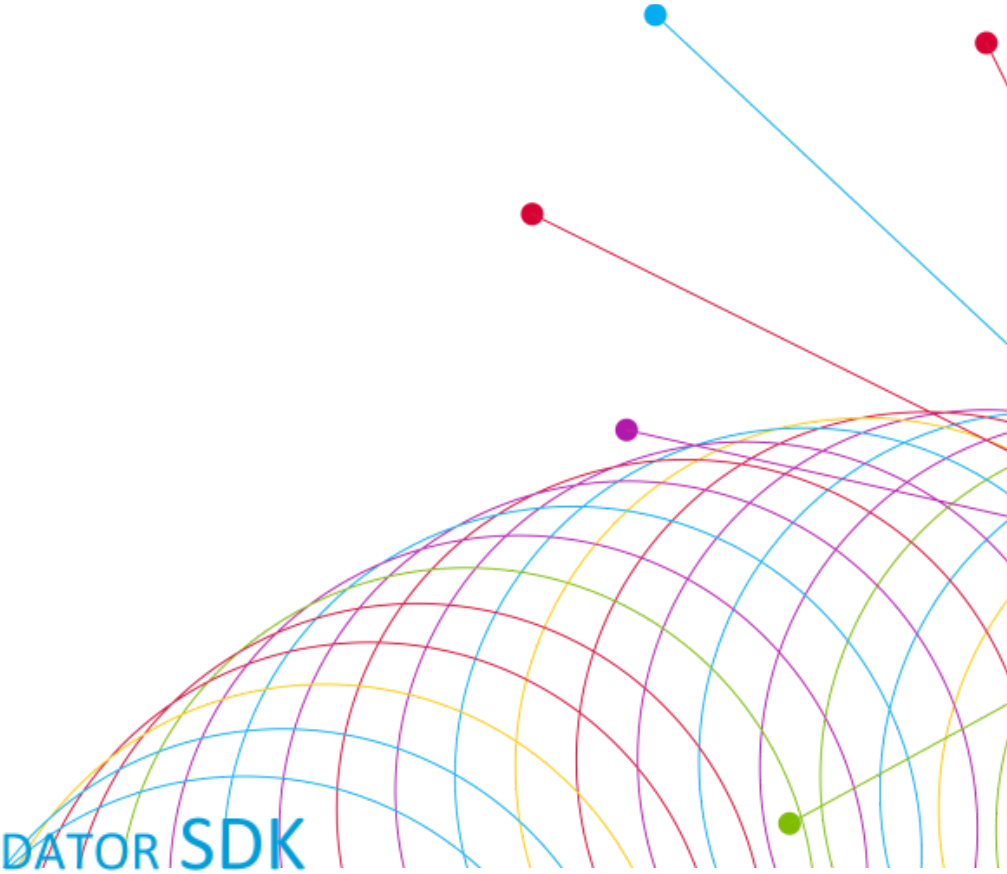

ID3 Tag Validator SDK

Version 1.7

Developer Guide

Detailed Revision History

Revision	Date	Description	Author(s)/Editor(s)
A	2014-08-05	Initial Draft	Lois Price / Lore Eargle
B	2015-05-27	Revised for release 1.5	Lois Price Lore Eargle (editor) Amy Gaither (editor)
C	2016-06-15	Revised for SDK release 1.7	Lois Price Lore Eargle (editor)

Contents

Summary 4
- Purpose of the SDK 4
- Audience for this User Guide 4
- Out of Scope 5
- Related Documents 5
- Customer Support 5

Architecture of the ID3 Tag Validator SDK 6
Quick Start to the ID3 Tag Validator SDK 7
Contents of the Nielsen ID3 Tag Validator SDK 8
The Software Interface: CId3TagValidator 11
- Overview of CId3TagValidator 11
- CId3TagValidator (The Constructor) 12
- ProcessData() 15
- GenerateSummaryReport() 15

ID3 Tag Validator Callback Class 15
- Overview 15
- Classes and Data Types Used by the Callback 16
- The Reported Test Results 22

The Sample Application 33
- NielsenId3TagValidator.cpp 34
- NielsenValidatorParameters (.cpp and .h) 34
- NielsenValidatorFileReader (.cpp and .h) 35
- NielsenValidatorLogger (.cpp and .h) 36

Running the Sample Application 37

List of Figures

Figure 1 – Architecture of the ID3 Tag Validator SDK 6
Figure 2 – ID3 Tag Validator Sample Application 33

List of Tables

Table 1 – ID3 Tag Validator SDK Contents 9
Table 2 – Constructor Arguments 13

Summary

Purpose of the SDK

As part of its strategy to measure audience viewing of content on consumer devices, Nielsen supports or endorses a variety of products that generate Nielsen ID3 tags and insert them into MPEG-2 transport streams. C/C++ developers may use Nielsen ID3 Tag Validator Software Development Kit (SDK) to create applications that monitor ID3 tags and diagnose errors in those tags.

These are just two of the possible uses of applications based on the Nielsen ID3 Tag Validator SDK:

- The Validator application may process a file of encrypted ID3 tags (for example, the output of the PCM-to-ID3 SDK sample application). In response, the Validator application delivers a summary report that includes a list of checks performed on the stream. Each check is assessed simply as <pass/fail/warning>.
- The Validator application may process a transport stream with embedded Nielsen ID3 tags. Again, the Validator application delivers a pass/fail summary report. In addition, the SDK delivers periodic reports on the “health” of the ID3 tags included in the stream, where the report interval is configurable.

The Validator SDK uses a set of callback functions within a C++ class to deliver both its periodic and summary reports. The callback functions give you (the developer) full control over the format of the reports and the method used to deliver them.

Audience for this User Guide

If you are an experienced C/C++ programmer, the Nielsen ID3 Tag Validator SDK allows you to create applications that analyze and evaluate Nielsen ID3 tag streams. This user guide provides descriptions of all functions exposed through the application programming interface (API) of the SDK software. Combined with the in-line comments in the sample application, this guide also provides examples illustrating how the API functions and callbacks are to be used.

The document assumes that you have working knowledge of the following:

- C/C++ software development on Microsoft® Windows operating system and/or the Linux® platforms
- The basics of Nielsen ID3 Tag generation and insertion
- The use of SDKs in developing end-user applications.

Out of Scope

This document does not provide detailed information about the format of Nielsen ID3 tags or about their role in audience measurement. For such information, please refer to the *Nielsen PCM to ID3 SDK Developer's Guide*.

This document concentrates on using the API in developing software applications. For a detailed description of the meaning of each of the thirty tests listed in the summary report and in periodic reports, please refer to the ID3 Tag Validator Application User Guide.

Related Documents

Apple. iOS Developer Library, "HTTP Live Streaming Metadata Spec." *https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/HTTP_Live_Streaming_Metadata_Spec/2/2.html*.

Nielsen ID3 Tag Validator Application User Guide. This manual describes in detail how to run the sample application and how to use it as a diagnostic tool.

Nielsen PCM-to-ID3 SDK Developer's Guide. This manual describes how to use the PCM-to-ID3 SDK to generate Nielsen ID3 tags. It provides an overview of the format of Nielsen ID3 tags and of their role in measuring audience viewing on a variety of consumer devices.

Customer Support

For technical support, please contact your Nielsen representative.

Architecture of the ID3 Tag Validator SDK

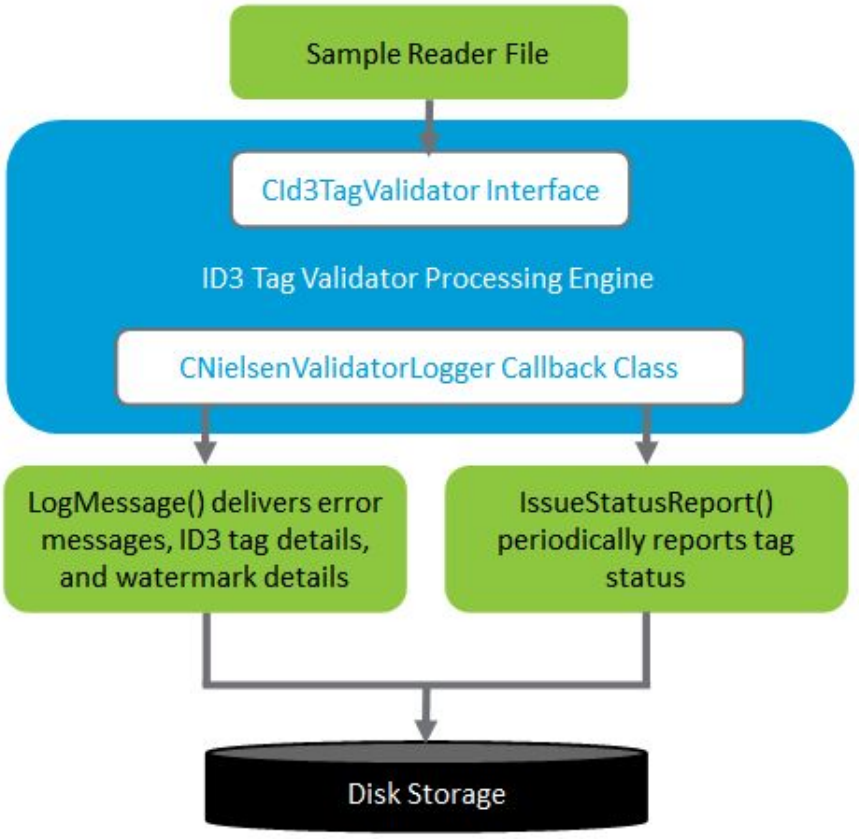

Figure 1 – Architecture of the ID3 Tag Validator SDK

Figure 1 illustrates the general architecture of the ID3 Tag Validator SDK.

- The dark blue box represents the ID3 Tag Validator Engine, which extracts ID3 tags from the input data and analyzes the tags. Periodically it issues status reports, and upon request, it delivers a summary report to the calling application.
- The light blue rectangle is the interface that developers use to deliver instructions to the SDK libraries.
- The green boxes represent source code for sample-application modules, provided as examples of how the SDK may be used.

Quick Start to the ID3 Tag Validator SDK

Classes

The ID3 Tag Validator SDK exposes two primary classes that you will use in creating your application:

- **CId3TagValidator** is the primary interface to the ID3 Tag Validator libraries. It exposes the functions that you use to deliver data to the SDK libraries and to generate a summary report.
- To enable the SDK to deliver reports and log messages to your application, you derive a class from the base class, **Id3TagValidatorLoggerBase**.

Create Your Application

1. Derive a class from **Id3TagValidatorLoggerBase**. Your class must implement these two functions:

   ```
   int LogMessage( int code, const char *message)
   ```

   The ID3 Tag Validator SDK invokes a call to LogMessage each time that it has an error or status message to deliver. If the code is a negative number, the message is an error message. If 'code' is set to 0, then the message is usually a status message. There are three other special settings of 'code':

   - 11111 indicates that 'message' is a detailed listing of a decrypted ID3 tag.
   - 22222 indicates that 'message' reports the fields of a NAES 2 or NAES 6 watermark extracted directly from the audio stream.
   - 12121 indicates that 'message' is a comma-delimited listing of a decrypted ID3 tag.

   ```
   int IssueStatusReport( ValidatorOutputFields_t *pValidatorStatus )
   ```

   The ID3 Tag Validator SDK invokes a call to IssueStatusReport each time that it has a periodic report to deliver. It delivers to the calling application a pointer to a structure that holds the following:

   - A list of .detected Nielsen SIDs.
   - The settings of fields in the ID3 INFO tag.
   - The results of up to 30 executed tests, each with an indication of pass/fail.

   Note that IssueStatusReport() is used to deliver both periodic and summary reports. The sample application source code delivered with the SDK provides an example of a handler for this callback.

2. Create an instance (*pValidatorLogger*) of the Logger class that you implemented.
3. Declare a variable (*nInputType*) that defines the input either as a stream of encrypted ID3 tags or as an MPEG-2 transport stream.

4. Declare a variable (*nPeriodicReport*) that specifies the type of output report(s):

   0 = Generate only the two required reports (Summary and Log)

   1 = Generate periodic reports, but no tag listings

   2 = Generate a detailed tag-listing, but no periodic report, no comma-delimited listing.

   3 = Generate both periodic reports AND a detailed tag listing, but no comma-delimited listing.

   4 = Generate a comma-delimited tag listing, but no periodic reports or detailed tag listings.

   5 = Generate a comma-delimited tag listing and periodic reports, but no detailed tag listing.

   6 = Generate a detailed tag listing and a comma-delimited listing, but no periodic reports.

   7 = Generate all possible reports.

5. If you elected to create periodic reports (options 1, 3 and 7), set *nPeriodicReportInterval* to the number of seconds between periodic reports.

6. Create an instance of CId3TagValidator, as follows:

```
pProcessor = new CId3TagValidator( nInputType,nPeriodicReportFlag,
nPeriodicReportInterval, pValidatorLogger );
```

7. Programmatically open a file or stream that holds the ID3 tags or transport stream.to be evaluated or monitored.

8. Repeatedly deliver blocks of data to *pProcessor*, using this call:

```
Processor->ProcessData( (uint8_t *)pData, nDataLength );
```

9. If you are processing a file, generate a summary report when the entire file has been processed:

```
pProcessor->GenerateSummaryReport();
```

10. Release all resources.

Contents of the Nielsen ID3 Tag Validator SDK

The SDK includes the header files, libraries, executables, and documents listed in Table 1. Each SDK package includes only the software components that apply to the platform/compiler configuration named in the package title. The subdirectory holding each of the components may vary slightly from the folder named in the table.

The SDK has been tested on Microsoft Windows 7 and Microsoft Windows 8 operating systems. Project files are included for Visual Studio® 2010, Visual Studio 2012, Visual Studio 2013 development systems. Other versions of Visual Studio may be available upon request.

Linux software has been tested on Fedora™ 19 (64-bit platform).

For Apple®, the SDK has been has been tested on Yosemite OS, 64-bit, compiled with the Clang compiler.

Table 1 – ID3 Tag Validator SDK Contents

Folder	Platform	Contents + (Description)
NielsenApp	Linux	NielsenId3TagValidator Each Linux package includes one executable, provided for demonstration/testing/debugging purposes.
Apps/Windows/x86/	Windows	NielsenId3TagValidator.exe (A 32-bit build of the sample application)
Apps /Windows/x64/	Windows	NielsenId3TagValidator.exe (A 64-bit build of the sample application)
docs	All	ID3 Tag Validator Application User Guide Nielsen ID3 Tag Validator SDK Release Notes ID3 Tag Validator SDK User Guide (this document) Validator_Readme.txt (legal notices)
inc	All	AC3InserterErrorCodes.h Id3TagValidator.h Id3TagValidatorLoggerBase.h IGenericValidatorApi.h ITagValidatorReader.h IValidatorStreamProcessor.h ValidatorOutputDefinitions.h w32stdint.h (Windows only) (Header files required to use the libraries in “lib” folder.) **Important** The contents of these header files must not be changed.
lib (or lib subdirectory – 32-bit Linux)	Linux	libCryptopp.a libDecodeAc3.a libId3TagAnalyzer.a libId3TagUtils.a libTsId3TagValidatorApi.a libTsProcessor.a (Static libraries that the sample application must link to, 32-bit version of Linux)
lib (or lib subdirectory – 64-bit Linux)	Linux	libCryptopp.a libDecodeAc3.a libId3TagAnalyzer.a libId3TagUtils.a libTsId3TagValidatorApi.a libTsProcessor.a (Static libraries that the sample application must link to, 64-bit version of Linux)

lib/Windows_NT/Win32/vs2010/dynamic lib/Windows_NT/Win32/vs2012/dynamic lib/Windows_NT/Win32/vs2013/dynamic	Windows	LibTsId3TagValidatorApi.lib (Static library that sample application must link to, for 32-bit platform, compiled with the corresponding version of Visual Studio, with dynamically linked runtime libraries)
lib/Windows_NT/Win32/vs2010/static lib/Windows_NT/Win32/vs2012/static lib/Windows_NT/Win32/vs2013/static		LibTsId3TagValidatorApi.lib (Static library that sample application must link to, for 32-bit platform, compiled with the corresponding version of Visual Studio, with statically linked runtime libraries)
lib/Windows_NT/x64/vs2010/dynamic lib/Windows_NT/x64/vs2012/dynamic lib/Windows_NT/x64/vs2013/dynamic		LibTsId3TagValidatorApi.lib (Static library that sample application must link to, for 64-bit platform, compiled with the corresponding version of Visual Studio, with dynamically linked runtime libraries)
lib/Windows_NT/x64/vs2010/static lib/Windows_NT/x64/vs2012/static lib/Windows_NT/x64/vs2013/static		LibTsId3TagValidatorApi.lib (Static library that sample application must link to, for 64-bit platform, compiled with the corresponding version of Visual Studio, with statically linked runtime libraries)
NielsenApp	All	NielsenId3TagValidator.cpp NielsenValidatorFileReader.cpp NielsenValidatorFileReader.h NielsenValidatorLogger.cpp NielsenValidatorLogger.h NielsenValidatorM3u8Reader.cpp NielsenValidatorM3u8Reader.h NielsenValidatorParameters.cpp NielsenValidatorParameters.h Makefile (Linux and Mac) NielsenId3TagValidator2010.sln (Windows only) NielsenId3TagValidtator2010.vcxproj (Windows only) NielsenId3TagValidator2012.sln (Windows only) NielsenId3TagValidator2012.vcxproj (Windows only) (Source code and project files required to compile the sample application whose executable is provided in the apps folder). **Note** Nielsen provides this source code as an example of how to use the SDK libraries. The developer may modify these files to meet the needs of his or her application and to include proper error checking.

The Software Interface: CId3TagValidator

Overview of CId3TagValidator

The class CId3TagValidator is the primary interface to the set of Nielsen ID3 Tag Validator SDK libraries. You may use this class to process, evaluate, and report on encrypted Nielsen ID3 tags delivered in any of these "containers":

- In buffers that hold ONLY encrypted ID3 tags, and no other data (for example, the output of the PCM-to-ID3 SDK callback function).
- In buffers that hold encrypted ID3 tags, intermingled with other data (for example, the log file created by the Nielsen ID3 tag sample application installed on tablet and phone devices).
- In buffers of transport-stream data, where the transport stream contains a metadata elementary stream that holds Nielsen ID3 tags. The metadata elementary stream must be properly declared in the PMT, as specified in "HTTP Live Streaming Metadata Specifications." (See the *References* section of this document).

Regardless of the "container" that carries the Nielsen ID3 tags, CId3TagValidator accepts and buffers blocks of data, then uses callback functions to deliver reports of its analysis.

CId3TagValidator is a very simple class. It exposes just three public functions: the constructor, ProcessData(), and GenerateSummaryReport(). These functions are described in the remainder of this section.

CId3TagValidator (The Constructor)

Use the CId3TagValidator constructor to do the following:

- Identify the type of stream to be processed (ID3 tag or transport stream)
- Specify which of the four optional callbacks you would like to receive. The optional callbacks provide data for these reports:
  - **Periodic reports**, which are used primarily with streaming (as opposed to file-based) input content. However, if the input stream is from a large file, periodic reports may be of some use in identifying specific problem spots in the file. The SDK libraries invoke the IssueStatusReport() function to deliver periodic reports at regular intervals.
  - **ID3 tag listing**, which is a listing of all settings in a single decrypted Nielsen ID3 tag. The SDK libraries invoke the LogMessage() function (with the 'code' parameter set to 11111) to deliver a listing of each ID3 tag encountered in the source stream.
  - **Watermark listing**, which is a listing of NAES 2 and NAES 6 watermarks found in AC-3 audio in the transport stream (if the input is a transport stream). The SDK libraries invoke the LogMessage() function (with the 'code' parameter set to 22222) to deliver a listing of each NAES 2 or NAES 6 watermark encountered in the source

AC-3 audio stream.

- **Comma-delimited tag listing:** which lists the key elements of a single ID3 tag in a single line with comma-delimited fields. The SDK libraries invoke the LogMessage() function (with the 'code' parameter set to 12121) to deliver a comma-delimited report of each ID3 tag encountered in the source stream.
- Define the interval between periodic reports (if periodic reporting is turned on)
- Identify the program to process, if the input is a multi-program transport stream. (Set to -1 if input is not MPTS).
- Register your implementation of the callback class (through which the SDK libraries deliver log messages and analysis reports to your application).

Data Types Used by the Constructor

The constructor uses two locally defined data types: ValidatorInputType_t and Id3TagValidatorLoggerBase. The definition for ValidatorInputType_t appears below. The definition of Id3TagValidatorLoggerBase appears on page .

```
// ValidatorInputType_t specifies the type of data that the
// ProcessData() call will receive (one buffer at a time).
// Selections are:
// * eTransportStreamType - ProcessData() receives buffers of MPEG-2
//         transport-stream
// * eRawTagFileType - ProcessData() receives buffers of data that include
//         raw (encrypted) Nielsen ID3 Tags.
typedef enum eValidatorInputType
{
    eDataTypeUnknown,
    eTransportStreamDataType,
    eRawTagDataType
} ValidatorInputType_t;
```

Other data types described in this section are defined in ValidatorOutputDefinitions.h.

Constructor Arguments

The constructor creates the resources required to carry out the analysis, and it configures components in accordance with the settings that it receives. Table 2 defines the constructor arguments.

Table 2 – Constructor Arguments

Parameter	Permissible Settings	Notes

`ValidatorInputType_t nInputType`	eTransportStreamDataType eRawTagDataType	Use this argument to specify whether your application will deliver buffers of transport stream, or buffers of a text stream that contains encrypted Nielsen ID3 tags.
`uint32_t nGeneratePeriodicReports`	0 – do not generate any of the three optional reports: periodic reports, detailed tag listings, or comma-delimited tag listings. 1 – Generate periodic reports, but not detailed or comma-delimited tag listings. 2 – Generate detailed tag listings, but not periodic reports or comma-delimited tag listings. 3 – Generate periodic reports and detailed tag listings. 4 – Generate comma-delimited tag listings, but no periodic reports or detailed tag listings. 5 – Generate periodic reports and comma-delimited tag listings only. 6 – Generate detailed tag listings and comma-delimited tag-listings, but no periodic reports. 7 – Generate all three optional report types.	Periodic reports are most useful when the incoming content is streaming (not read from a file). However, with very long files, periodic reports may be useful in identifying “problem spots” in the content. Tag listings are most useful for diagnostic purposes. Comma-delimited tag listings do not provide all of the tag details, but they are easy to review in spreadsheet format. Note that log messages are not optional – they are always generated. The summary report is always available post-processing, even if none of the three optional reports is selected.
`uint32_t nPeriodicReportIntervalInSeconds`	Recommended setting is 30 seconds or greater.	If the periodic-reports flag is 1, this value specifies the number of seconds between periodic reports.
`int32_t nSelectedProgram`	-1 for SPTS > 0 for MPTS	Specifies the program number to process in a multiprogram transport stream
`Id3TagValidatorLoggerBase *pLogger`	Points to a callback class object	It is your responsibility to implement an object of type Id3TagValidatorLoggerBase for the purpose of receiving and handling log messages, tag listings, and ID3 Tag analysis reports.

ProcessData()

The ProcessData function accepts and processes a block of input data (either transport-stream data or text content including encrypted ID3 tags). It buffers the incoming data, then delivers the buffer to a transport-stream processor or directly to an ID3 tag processor for parsing and analysis. The function accepts a pointer to a buffer of incoming data, followed by an argument specifying the number of bytes in the buffer. The function returns 0 if it completes successfully, or a negative value if it encounters an error. The function declaration:

```
int ProcessData( uint8_t *pData, uint32_t nSize );
```

GenerateSummaryReport()

Your application must call GenerateSummaryReport()to generate the final report, after all incoming content has been processed. After you call this function, the IssueStatusReport() callback delivers the report to your application. The function returns 0 if and only if it finishes successfully.

ID3 Tag Validator Callback Class

Overview

The ID3 Tag Validator SDK libraries deliver error and status messages as well as analysis reports by invoking calls to public methods of a callback class. It is your responsibility as a developer:

- To define a class derived from Id3TagValidatorLoggerBase.
- To implement the LogMessage() function to handle messages in a way that meets the needs of your application. Be aware that the SDK delivers special message types (as described above) by setting the code to 11111, 22222, or 12121. You may assume that all other LogMessage calls (with code <= 0) are status or error logs.
- To implement the IssueStatusReport() function to receive reports in a way that meets the needs of your application.
- To create a callback object, based on the class that you created, and to pass a pointer to that object as the last argument of the CId3TagValidator constructor.

Classes and Data Types Used by the Callback

The Base Class: Id3TagValidatorLoggerBase

The callback base class is defined on the next page. The object that you create must implement both the LogMessage and the IssueStatusReport functions.

```
class Id3TagValidatorLoggerBase
{
public:
    //! Special code used for debugging tag details
    static const int cTagReportCode = 11111;

    //! Special code used to report lines in comma-delimited file.
    static const int cCommaDelimitedReportCode = 12121;

    //! Special code used for debugging watermark details
    static const int cWatermarkReportCode = 22222;

    * \brief      Default constructor
    Id3TagValidatorLoggerBase( void ){}

    * \brief      Default destructor
    virtual ~Id3TagValidatorLoggerBase(void){}

    * \brief      Delivers a status or error code and a detailed message.
    virtual int LogMessage( int code, const char *message) = 0;

    * \brief      Issues a report of the current status of the ID3-tagging
    process.
    virtual int IssueStatusReport(ValidatorOutputFields_t *pValidatorStatus) =
    0;
};
```

Report Structure: ValidatorOutputFields_t

Both the periodic and summary reports are delivered in the ValidatorOutputFields_t structure, defined below.

```
typedef struct structValidatorOutputFields
    {
    ValidatorReportType_t nReportType;
```

```
    uint32_t nTagTime;
    uint32_t nDuration;
    uint32_t nBreakoutType;
    ValidatorSidInfo_t SidList[_MAX_VALIDATOR_SIDS_];
    uint16_t nSidCount;
    ValidatorInfoTagValues_t InfoTagSettings;
    ValidatorCheckData_t StatusCheckList[eTotalValidatorChecks];
    } ValidatorOutputFields_t;
```

nReportType

The nReportType setting specifies whether this is a periodic or summary report. It is of type ValidatorReportType_t, defined below. Note that periodic reports are delivered only if the constructor is configured with nGeneratePeriodicReports set to 1 or 3.

```
//! Type of report: periodic or summary
typedef enum eValidatorReportType
{
    eValidatorPeriodicReport,
    eValidatorSummaryReport,
    eValidatorReportTypeUnknown
} ValidatorReportType_t;
```

nTagTime

In a summary report, the nTagTime field is set to the “end” time of the last tag processed. In a periodic report, the nTagTime is set to the start time of the most recently processed ID3 tag. This field is used strictly as a “label” for the report, to distinguish it from earlier and subsequent reports.

nDuration

For periodic reports, the nDuration setting represents the number of seconds since the last periodic report (or, for the first periodic report, the number of seconds since the beginning of the content).

For summary reports, nDuration specifies approximate length, in seconds, of all of the processed content.

The calculation of the duration is based on the first and last tag times. For example, for the summary report, the duration is the difference between the “last” tag time of the most recent tag and the “first” tag time of the first decoded tag. If the tag times are in error the duration, too, is in error.

nBreakoutType

This is a value between 0 and 99 that represents the breakout type of this content. See Appendix – Breakout Codes on page for breakout-type settings that are currently in effect. Note that this field is now ignored by most downstream processes.

SidList

All reports delivered by IssueStatusReport() include a list of Nielsen SIDs that were included in the processed tags. For summary reports, the SidList contains a list of all SIDs found during the processing of the entire content. For periodic reports, the SidList holds only the SIDs that were discovered during the current interval.

Each element of the SidList is defined as:

```
//! strValidatorSidInfo defines a single record in the SID list
//! Each record holds the SID value, the number of SIDs with that
//! value recorded during the report period, and the level (PC/FD)
//! of the watermark from which the SID was extracted.
typedef struct strValidatorSidInfo
{
    uint32_t nSid;  // 16-bit SID
    uint32_t nNaes2SidCount; // Number of EDUs with this NAES 2 SID
    uint32_t nNaes6SidCount; // Number of EDUs with this NAES 6 SID
    ValidatorSidLevel_t nLevel; // specifies PC or FD (watermark level)
} ValidatorSidInfo_t;
```

nSidCount

The nSidCount setting specifies the number of entries in the SidList.

InfoTagSettings

This structure holds the settings of the INFO tag, which remain constant throughout an entire piece of content. The user of your application may review these settings for validity. Please check the PCM-to-ID3 SDK for proper settings of these fields.

```
typedef struct strValidatorInfoTagValues
{
    //! Unique ID assigned by Nielsen. Three digits + null-terminator
    uint8_t     vendor_id[4];

    //! 0 = reserved, 1 = in-home streaming devices, 2 = transcoder,
    //! 3 = segmenter.
```

```
    uint8_t     device_type;

    //! 16-byte unique device ID. Printable characters
    uint8_t     device_id[17];

    //! 4-byte system version, plus null-terminator
    uint8_t     system_version[5];

    //! 1-byte version of the Nielsen ID3 tag SDK
    //! Major version = high-order nibble; minor version = low-order nibble
    uint8_t        sdk_version;

    //! 16-byte ASCII text field holding codec name & bit rate, no spaces.
    //! Examples: "AC3-5.1,384kbps", "AAC-HE,192bps". No longer required.
    uint8_t     audio_codec[17];

    //! 48-byte distributor ID. Usually a URL beginning with “www”
    uint8_t        distributor_id[49];

} ValidatorInfoTagValues_t;
```

StatusCheckList

The status checklist is an array of elements that identify the “status” (pass/fail/warning) of a list of tests. Each element in the array represents a single test.

```
//! Information supplied with each "check" report
typedef struct strValidatorCheckData
{
    //! Recommended label for this test
    char strTestLabel[80];

    //! Values: eNoStatus, eValidatorSuccess,
    //! eValidatorWarning, eValidatorFailure
    ValidatorOutputStatus_t nStatus;

    //! Number of events or errors that were counted
    //! for this "check" field
    uint32_t nCount;

    //! If we are checking an "event" that must occur,
```

```
    //! this is the minimum that we must see for the
    //! check to pass (for the given period of time).
    uint32_t nMin;

    //! If we are checking a total number of errors,
    //! this is the maximum number that may occur
    //! in order for the check to pass.
    uint32_t nMax;

} ValidatorCheckData_t ;
```

Each element of the StatusCheckList represents either an “event” that must occur or an “error.”

Following the nStatus field are three values:

- **nCount** – the number of times that the event or error occurred during the period (for periodic report) or during the entire content (for summary report).
- **nMin** – if the field represents an “event,” nMin specifies the minimum number of times the event must occur in order for the test to “pass.”
- **nMax** – if the field represents an “error,” nMax specifies the maximum number of times that the error may occur before the test is considered to “fail.”

Altogether 30 tests are represented in the StatusCheckList. Each test is represented by the enum value of the same position. The enum values appear on the next page. The tests themselves are described thoroughly in “The Reported Test Results” on page , with even more details provided in the *ID3 Tag Application User Guide*.

```
//! These are the checks performed, in the order that
//! they occur in the StatusCheckList array.
typedef enum eValidatorCheckCategories
{
    eDataTagCheck, // Based on number of DATA tags counted
    eTagFormatCheck, // Tag does not have slashes in correct places
    eSequenceNumberCheck, // Based on number of missing tag sequence numbers
    eTimeGapCheck, // Based on number of time-gaps between tags
    eTagDurationCheck, // Based on duration of tag (first time to last time)
    eCidSidCheck, // Based on value of CID SID compared to EDUs
    eCidTsCheck, // Check that TS is set to most recently passed 3 AM
    eCidTsOffsetCheck, // Based on value of CID TS + Offset, compared to EDU
    eBreakoutTypeCheck, // Reports change in breakout code
    eWatermarkTimeCheck, // Based on watermark time-code errors
```

```
        eWatermarkLevelCheck, // Based on changing watermark level for a SID
        eInfoTagCheck, // Based on number of INFO tags counted
        eInfoTagIntervalCheck, // INFO tag interval outside of 290-310-second
        range?
        eITCidError, // Either 1 or both of the INFO tag CIDs are not set to 0
        eITOffsetError, // 1 or both of INFO tag Seconds-Since are not set to 0.
        eITVendorIdError, // Invalid or missing vendor ID in INFO tag
        eITDeviceTypeError, // Invalid or missing device type in INFO tag
        eITDeviceIdError, // Invalid or missing device ID in INFO tag
        eITSystemVersionError, // Invalid or missing version number in INFO tag
        eITSdkVersionError, // Invalid or missing SDK version number in INFO tag
        eITDistributorIdError, // Invalid or missing distributor ID in INFO tag
        eTagPlacementCheck, // Based on synchronization of tag with audio codes
        eTagTimeCheck, // Based on synchronization of PTS and tag times
        eId3PesPacketCheck, // Counts number of PES-packet headers.
        ePesOwnerIdCheck, // Based on presence of 'www.nielsen.com' within tag
        eMetadataPesPtsCheck, // Based on presence of PTS in the tag PES header
        ePesStreamTypeCheck, // based on correct stream type (0xbd) in PES header
        eCompleteTagPayloadCheck, // based on complete tag present in PES packet
        ePmtDescriptorCheck, // Based on presence of ID3 Tag descriptor in PMT
        loop
        ePmtElemStreamCheck, // Based on ID3-tag elementary stream type in PMT
        loop
        eTotalValidatorChecks //30
} ValidatorCheckCategories_t;
```

The Reported Test Results

Overview

Note The previous section, "StatusCheckList," describes the array that holds the 30 tests discussed below.

Although IssueStatusReports() uses the same structure to deliver periodic reports and the summary report, the meaning of each test or the criteria for passing the test may differ based on the report type. In all cases, however, the scope of the test for periodic reports is limited to the interval that the report covers. For the summary report, the test applies to all of the content that was processed, from beginning to end.

Each of the tests returns a status of *pass/fail* (or, in some cases, *warning* or *no status*). The pass/fail status is simply a quick indicator of the overall status of the stream with respect to the test in question. The <count/min/max> fields provide more information, often helping to assess the severity of the problem if the status is reported as "fail." Finally, for error test failures, the status messages delivered via LogMessage() provide specific information about each incident that contributed to the error-count.

In the examples below, the results of each test are represented as **[Pass | Fail | Warning | No Status] < count / min / max>** , where:

- *Pass/fail*, obviously, indicates whether the ID3 tag stream passed or failed the test for the period in-scope.
- *Warning* is typically issued when the test itself provides background information, but does not contribute to the overall validity of the tags themselves.
- *No status* indicates that the test does not apply to this stream or to this interval of the stream.
- *Count* indicates the number of events or errors counted for the scope of the report
- *Min* for an "event" check, indicates the minimum number of event occurrences for the test to pass. For an "error" check, *min* is set to *0*.
- *Max* for an "error" check, indicates the maximum number of errors tolerated before the test is considered a failure. For an "event" check, *max* is set to *0*.

If the status of a test is reported as *fail*, the person using your application should take these steps:

- Note the **degree** to which the processed data failed the test by comparing the count to the min or max value (whichever applies). Especially for a periodic report with a short interval, a single error may be enough to trigger a report of *fail*. Close analysis may reveal the failure report is inconsequential because the report interval was so short.
- Check the log messages to determine the **reason** for the failure report. If the log message includes a time or sequence number, you may be able to match specific log messages to the *count* value reported. On the following pages, typical log messages are listed for each test. **Be aware that the wording of the message strings may change in future versions of the SDK; however, the corresponding error codes**

will not change.

- Refer to the *ID3 Tag Application User Guide* for more detailed information pertaining to each of the thirty checks.

Test 1: DATA-Tag Check (eDataTagCheck)

For periodic reports and the summary report, the DATA-tag check indicates the number of data tags detected for the period in-scope. That value is compared to the *min* value to determine the pass/fail status of the test.

Ideally, DATA tags arrive at 10-second intervals. However, in the presence of many watermarks, the tags may arrive more frequently.

Example

Pass <599/512/0>

In this case, 599 DATA tags were detected, compared to 512 required for the test to pass.

Test 2: Tag-Format Check (eTagFormatCheck)

The Validator SDK inspects the placement of the first three forward slash ('/') characters after the *www.nielsen.com* string in the Nielsen ID3 tag. It also verifies the correct placement of the last '/' characters in the string. This second check helps to determine the presence of both transport-packet payloads that comprise a single ID3 tag. The Validator SDK declares a tag-format error if one or more '/' characters are missing or out of place.

The Validator SDK logs an error, too, if the spacing between PES headers in an incoming transport stream indicates that the tag-length may be incorrect.

Note that Validator SDK versions prior to 1.7 sometimes declared tag-format errors if there were non-Nielsen elements in the metadata PES (Packetized Elementary Stream). Version 1.7 logs instances of non-Nielsen ID3 tags, but does not count them as tag-format errors.

For periodic and summary reports, the tag-format check compares the number of tag-format errors to the maximum number of errors permitted for the period.

Corresponding Log Messages

All of these log messages, which include an error code of `_VALIDATOR_TAG_FORMAT_ERROR_` (-51), are indications of tag-format errors.

```
Tag error: Separators missing.

Tag Error: Detected corrupted, partial, or non-Nielsen tag.

Tag Error: Error decoding ID3 tag.

Corrupted tag (slash error): Sequence Number xx, Tag Time xx.

Corrupted tag (checksum error): Sequence Number xx, Tag Time xx.
```

Example

Pass <1/0/30>

There was 1 error, but up to 30 errors would have been permitted.

Test 3: Sequence-Number Check (eSequenceNumberCheck)

The sequence-number check indicates the number of that the Validator SDK detects an out-of-order or missing sequence number during the report interval. For example, if the Validator SDK detects that tag 102 follows tag 100, it reports a sequence number error because tag 101 is missing.

Note that INFO tags sometimes occur out of order, especially at the beginning of the stream. As long as the INFO tags are not missing from the stream altogether, it is acceptable for INFO tags to fall slightly out of order.

If a facility uses a fail-over tag-inserter system, the tag sequence numbers of the primary and fail-over system may not be synchronized. However, the tag times and watermark times of the two systems should be close. At the point of the fail-over, there may be a tag overlap or a tag gap of up to 9 seconds. If there are sequence number errors, but no tag-gap errors, assume the sequence-number errors are the result of switching to a fail-over system and ignore these errors.

Corresponding Log Message

The error code `_VALIDATOR_SEQUENCE_NUMBER_ERROR_ (-52)` indicates that a sequence number is missing or out-of-order. This error is generated whenever the sequence number of the current tag is not one greater than the sequence number of the previous tag.

These log messages may accompany the -52 error code:

```
Seqno Error and Gap in Tag Times (xx seconds): Sequence Number xx, Tag Time
xx (time string)
```

```
Seqno Error, No Time Gap: Sequence Number xx, Tag Time xx (time_string)
```

Example

Pass <1/0/2>

There was 1 missing sequence number for the in-scope period; however, up to 2 errors would have been tolerated.

Test 4: Time-Gap Check (eTimeGapCheck)

Under ideal conditions, each DATA tag begins when the previous tag ends. A notable exception to this rule occurs when a tag stream includes ID3 tags from two different tag-generators (a primary device and a fail-over device). If the tag start/stop times indicate a time-overlap or a time-gap of more than 9 seconds between tags, Validator SDK reports a time-gap error.

Corresponding Log Message

The following log message, with an error code of `_VALIDATOR_SEQUENCE_NUMBER_ERROR_ (-52)` appears if the time gap is accompanied by a missing sequence number:

```
Seqno Error and Gap in Tag times (xx seconds): Sequence Number xx,
Tag Time xx.
```

Error code `_VALIDATOR_TAG_OVERLAP_ERROR_ (`-74) indicates that two consecutive ID3 tags overlap.

Example

Pass <0/0/2>

No gaps in time occurred between consecutive tags, but up to 2 such gaps would have been permissible.

Test 5: Tag-Duration Check (eTagDurationCheck)

Under ideal conditions, a DATA tag spans 10 seconds. However, if 10 watermarks arrive in less than 10 seconds, the tag-duration is shorter; in this case, a tag shorter than 10 seconds is not considered to be faulty. Regardless of the number of watermarks (or EDUs), any tag that is longer than 12 seconds or shorter than 4 seconds is reported to have a tag-duration error.

Corresponding Log Message

The following log messages, with error code `_VALIDATOR_DURATION_CHECK_ERROR_ (-54),` are indications of tags that are too short or too long:

```
Tag time error, sequence number xx: Tag is short, but EDU not filled.

Tag time error, sequence number xx: Tag duration is less than 4 seconds.

Tag time error, sequence number xx: Tag duration is greater than 12 seconds.
```

Example

Pass <1/0/3>

Although only 1 tag was shorter than 4 seconds or longer than 12 seconds, it would have been permissible for there to have been up to 3 such tag-length errors before the test was labeled “fail.”

Tests 6 – 8: Invalid CID: CID SID Error, CID TS Error, CID TS Offset Error

There are three ways that a CID (Content Identifier) may be in error:

- The wrong PC or FD SID may have been selected (eCidSidCheck)
- For live-streaming content, the CID time stamp may have been set to a value other than 3 AM, usually indicating a daylight-saving-time error (eCidTsCheck).

- The PC or FD CID timestamp + offset may not identify the correct ID3 tag. (eCidTsOffsetCheck)

Note that if the CID of a specific ID3 tag is invalid for more than one reason, it may be that only one error is tallied in the StatusCheckList. Please refer to the application user's guide for more detailed information pertaining to these three tests.

Corresponding Log Message

There are quite a few error messages that may be logged in the case of a CID error of any kind. All of the errors are tagged with one of the following error codes. The message string itself indicates whether the error is due to an invalid SID, timestamp, or tag offset.

```
_VALIDATOR_PC_CID_ERROR_  -55
_VALIDATOR_FD_CID_ERROR_  -56
_VALIDATOR_BOTH_CID_ERROR_ -57
```

Example

ID3 Tag: CID SID Test: Pass <0/0/6>

ID3 Tag: CID Timestamp Test: Pass <0/0/6>

ID3 Tag: CID TS Offset Test: Pass <1/0/6>

This example shows the following:

- No PC or FD CID errors due to invalid SID (up to 6 would have been permitted)
- No PC or FD CID errors due to timestamp (up to 6 would have been permitted)
- One PC or FD tag-offset error due to offset error (up to 6 would have been permitted)

Test 9: Breakout Type Check (eBreakoutTypeCheck)

The SDK reports an error if the breakout type falls outside the range "00" and "99." It logs an informational message if the breakout type changes, but does not declare it as an error. Note that Nielsen no longer uses the ID3 tag breakout code.

Corresponding Log Message

This message is logged if the breakout type is out of range ("00" to "99"), along with error code `_VALIDATOR_BREAKOUT_TYPE_ERROR_ (-58)`:

```
Invalid breakout code: Sequence Number xx, Tag Time xx (time string).
```

This message is logged whenever the breakout code changes, along with error code `_VALIDATOR_BREAKOUT_TYPE_ERROR_ (-58)`:

```
Breakout type changed from xx to xx: : Sequence Number xx, Tag Time xx.
```

Example

Pass <0/0/1>

Although there were no breakout errors, the SDK would have allowed one error before declaring that the test was labeled “fail.”

Tests 10 and 11: Watermark Time and Level Checks

The Validator SDK keeps track of any ID3 tag EDUs (watermarks) that have time codes that appear to jump forward or backward at a pace that is not synchronized with the running transport-stream or tag “clock.” The SDK also keeps track of changes to the daylight saving time (DST) flag setting. Any discrepancies between the expected watermark time (or DST flag) and the actual setting are tallied as watermark time warnings.

Likewise, the SDK keeps track of any watermarks for which the watermark level of any given SID has changed (for example, from PC to FD). Such a change could explain a resulting CID error. The SDK records this error in the eWatermarkLevelCheck element of the StatusCheckList array.

These observations regarding watermarks do not necessarily signify errors. Instead, they provide supporting information if there are reported CID errors in a specific ID3 tag. The watermark warnings should be considered informational only.

Corresponding Log Message

All log messages pertaining to watermark time variations and level changes are marked by log codes `_VALIDATOR_WATERMARK_TIME_WARNING_(-59) and _VALIDATOR_WATERMARK_LEVEL_WARNING_ (-60)`. The corresponding message strings provide detailed information that may prove useful in classifying / analyzing CID errors and warnings.

Example

Pass <4/0/22>

Although there are 4 watermark time-variations, up to 22 would be permitted. The 4 instances might provide useful background information pertaining to CID settings.

Test 12: INFO-Tag Check (eInfoTagCheck)

Periodic Report

For a periodic report, the INFO-tag check monitors the number of seconds since the last INFO tag was detected in the stream. Within the first 300 seconds of content, if no INFO tags have yet been seen, the INFO-tag check indicates the number of seconds since the start of content.

INFO tags are expected to arrive at 5-minute intervals. If 310 seconds expire without an INFO tag, the INFO-tag check is considered to have failed for that period.

Example

Pass <68/0/310>

This example indicates that 68 seconds have passed since the last INFO tag. The status is “pass” because 310 seconds have not yet transpired.

Summary Report

For a summary report, the INFO-tag check reports the number of INFO tags that occurred during the entire length of processed content, as compared to the minimum number of INFO tags that must be seen in order for the test to pass.

Example

Pass <19/16/0>

In this case, 19 INFO tags were detected, exceeding the minimum requirement of 16 tags.

Test 13: INFO-Tag Interval Check (eInfoTagIntervalCheck)

In that both tests report on the occurrence of INFO tags, the INFO-tag interval check is related to the INFO-tag check. However, the *eInfoTagIntervalCheck* is an *error* report, not an *event* report. It indicates the number of times an INFO tag appeared less than 290 seconds or more than 310 seconds after the last INFO tag. When an INFO tag is reported as missing, the next tag may also be reported as having “arrived too soon.” The result is that two errors are logged, although only one error occurred.

Corresponding Log Messages

If INFO tags arrive too soon or too late, the error code `_VALIDATOR_TIME_GAP_ERROR_ (-53)` precedes a message like this:

```
INFO Tag is missing or late: sequence number xx
```

Example

Pass <0/0/0>

Because of a short report-interval in the example above, no errors would have been tolerated. However, because no errors occurred, the test passed.

Tests 14 and 15: INFO Tag Errors (eITCidError and eITOffsetError)

For INFO tags, both the PC (Program Content) and FD (Final Distributor) CIDs should be set to *X100zdCIGellgZnkYj6UvQ==,* representing a SID of 0 and a timestamp of 0. If either of the CIDs is set to any other value, Validator SDK logs a CID SID error.

Likewise, the tag offsets for all INFO tags should be set to 0. If any INFO-tag offsets are not set to 0, the SDK logs a CID offset error.

Corresponding Log Message

Log messages accompany all detected INFO-tag CID and offset errors of the types described above. The error codes for these log messages are `_VALIDATOR_INFO_CID_ERROR_ (-61)` and `_VALIDATOR_INFO_OFFSET_ERROR_ (-62)`.

Example

Pass <0/0/1>

This report indicates that, although there are no INFO-tag CID (or INFO-tag offset) errors, one such error would have been acceptable for this period of time.

Tests 16 – 21: INFO Tag Field Settings

The SDK keeps a tally of errors that occur in each of six INFO-tag fields, excluding the audio CODEC setting, which is no longer required. For each of the six fields, the SDK registers any field with a missing or invalid setting. The length and format of each of the fields is defined in the *Nielsen PCM-to-ID3 SDK Developer's Guide*.

In most cases, the SDK simply verifies that the field is filled and that it does not have any characters considered to be "illegal" for that field.

Because all INFO tags are identical, the SDK does not log a message for each observed error. Instead, in the periodic and summary reports, the IssueStatusReport() callback provides the settings of all INFO fields.

These are the StatusCheckList fields that hold the INFO-tag reports:

```
eITVendorIdError, // Invalid or missing vendor ID in INFO tag

eITDeviceTypeError, // Invalid or missing device type in INFO tag

eITDeviceIdError, // Invalid or missing device ID in INFO tag

eITSystemVersionError, // Invalid or missing version number in INFO tag

eITSdkVersionError, // Invalid or missing SDK version number in INFO tag

eITDistributorIdError, // Invalid or missing distributor ID in INFO tag
```

Test 22: Tag Placement Check (eTagPlacementCheck)

If the incoming content is a transport stream that includes a watermarked AC-3 elementary stream, the tag-placement error field keeps track of the number times that an ID3 tag appears to be out-of-sync with the audio content that it represents. If the incoming stream is not a transport stream or if the audio format is not AC-3, the status is set to "No Status."

Corresponding Log Message

If the ID3 Tag Validator analyzer determines that a tag is badly placed in the transport stream, it issues this log message, using the `_VALIDATOR_TAG_PLACEMENT_ERROR_ (-64)`:

```
Tag is xx seconds separated from audio watermarks at sequence number xx, tag
time xx.
```

Example

Pass <0/0/1>

This report indicates that, although there are no tag-placement errors, one such error would have been acceptable for this period of time.

Test 23: Tag Time Check (eTagTimeCheck)

The Validator SDK compares the rate of advance of the metadata Presentation Time Stamp (PTS) to the rate of advance of the first or last tag time. The PTS is the MPEG-2 feature that synchronizes the component elementary streams of the transport stream. If the tag times advance at a rate that is substantially different from the rate at which the PTS is advancing, then the SDK declares a tag-time error.

Corresponding Log Message

If the ID3 Tag Validator analyzer determines that a tag is not synchronized with the audio PTS, it issues a log message using the `_VALIDATOR_TAG_PLACEMENT_ERROR_ (-64)`:

```
Tag time out-of-sync with PTS.  Delta = xx.
```

Example

Pass <0/0/5>

This report indicates that, although there are no tag-time errors, 5 such error would have been acceptable for this period of time.

Test 24: PES Packet Header Check (eId3PesPacketCheck)

The ID3 Tag Validator counts the number of PES-packet headers that are associated with the Nielsen ID3 tag metadata stream. There should be at least one of these PES packet headers every 10 seconds. If the PES headers are missing (even though the PMT lists the stream in its elementary loop) it is possible that the PES stream itself was omitted from the transport stream during multiplexing process.

This field is an “event” count, not an error count. The Validator SDK does not issue log messages for this event.

Example

Pass <69/53/0>

This report shows that 69 Nielsen PES headers were found. At least 53 instances are required, so the test passed.

Test 25: PES Owner-ID Check (ePesOwnerIdCheck)

When ID3 Tag Validator finds the string “ www.nielsen.com” near the beginning of a PES packet, it logs the event. The purpose of this test is to ascertain that the ID3 tag metadata elementary stream is, indeed, a *Nielsen* ID3 tag stream. Just one Nielsen URL event is needed to confirm this fact.

Because this is an “event” check and not an error check, no message is logged when a Nielsen owner-ID is detected.

Example

Pass <69/1/0>

This report shows that 69 Nielsen owner-IDs were found. Only one owner-ID is required to declare this a Nielsen ID3 tag. Therefore, the test passed.

Test 26: Metadata PES PTS Check (eMetadataPesPtsCheck)

In order for ID3 tag packets to be multiplexed properly into a transport stream, each PES header must contain a PTS. Each time the PTS is missing from a Nielsen metadata header, the Validator SDK increments the eMetadataPesPtsCheck field in the StatusCheckList array.

Corresponding Log Message

If the ID3 Tag Validator analyzer determines that the PTS field is missing from the Nielsen ID3 tag PES header, it logs an error message using the `_VALIDATOR_PES_PTS_ERROR_ (-67)` code. The corresponding message is.

```
Missing PTS in Nielsen PES header.
```

Example

Pass <0/0/4>

This report shows that no PTS fields were missing from the PES header. However, up to four missing PTS fields would have been tolerated. As a result, this check passes.

Test 27: PES Stream Type Check (eCompleteTagPayloadCheck)

Each Nielsen metadata PES packet header must have a stream ID of 0xBD, and it must include a non-zero PES packet size, as specified in the document “Timed Metadata for HTTP Live Streaming.” If Validator SDK parses a Nielsen ID3-tag PES packet for which one or both of these conditions is not true, it logs a stream-type error.

Corresponding Log Message

If the PES packet stream ID is set incorrectly or if the PES packet size is set to 0, one of the following error messages is sent with a code of -3:

```
PES header not valid for ID3 Tag Metadata Packet.

PES packet length is set to 0.
```

Example

Pass <0/0/5>

This report shows that there were no PES header errors found.

Test 28: Complete Tag Payload Check (eCompleteTagPayloadCheck)

If the size of the payload in the Nielsen ID3 metadata stream is not large enough to hold a complete Nielsen ID3 tag the Validator adds to the tally of incomplete ID3 tags. An incomplete payload cannot be decoded.

Each Nielsen ID3 tag spans two transport packets. If the size of the payload (as specified in the PES header) is correct, but the second transport packet contributing to the payload is missing, Validator logs an incomplete-tag error.

Corresponding Log Message

If the ID3 Tag Validator analyzer determines that the metadata PES payload is the wrong size, it logs this error message, using the `_VALIDATOR_PES_PACKET_ERROR_ (-65)` code:

```
Invalid tag returned from tag buffer.  Wrong size.
```

Example

Pass <0/0/2>

This report shows that no PES payloads were the wrong size. However, up to two badly sized PES packets are tolerated. As a result, this check passes.

Tests 29 and 30: PMT Elementary Stream and Descriptor Checks (ePmtElemstreamCheck and ePmtDescriptorCheck)

In order for an MPEG-2 decoder to detect a Nielsen ID3 tag elementary stream, it must find an entry in the PMT elementary-stream loop for a stream of type 0x15, followed by an MPEG-2 ID3 tag descriptor.

- Each time that it finds a stream of type 0x15 in the loop, the Validator increments the ePmtElemStreamCheck tally.
- Each time that it finds an ID3 tag descriptor, the Validator increments the ePmtDescriptorCheck.

Because both of these checks are “event” checks, not error checks, no messages are logged when the tallies are incremented.

For more information regarding the proper formation of PMT packets for ID3 Tag Metadata, refer to “HTTP Live Streaming Metadata”.

Example

Pass <7102/5/0>

This report shows that the proper descriptor was included in thousands of PMTs, when only 5 instances are required. Obviously, the test passed.

The Sample Application

The sample application provided with the SDK reads data from file and delivers it in buffers to the SDK libraries for processing and analysis. You can find the source code for this sample application in the NielsenApp folder of the SDK package. Although not production-ready, the source code does include helpful in-line comments. You may include the code, in whole or in part, in your own application. However, you may need to add error checking or customized settings to meet the needs of your application. Feel free to modify it to meet your needs, **except for the header files in the inc folder**, which are tightly linked to the SDK libraries and may not be edited.

In Figure 2, the blue rectangles represent classes and files whose source code is provided as part of the Nielsen ID3 Tag Validator SDK.

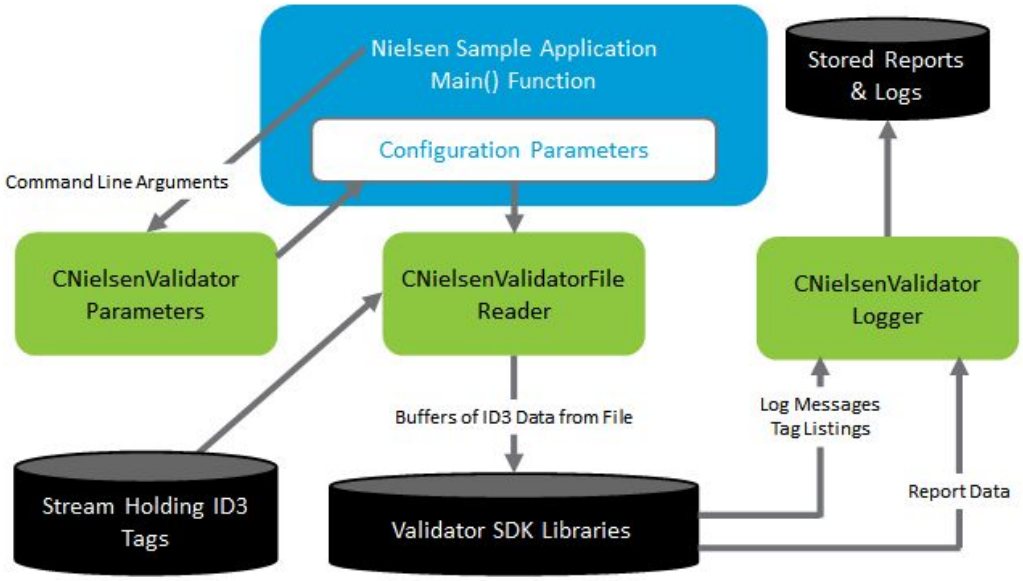

Figure 2 – ID3 Tag Validator Sample Application

NielsenId3TagValidator.cpp

NielsenId3TagValidator.cpp defines the main() function, which performs these basic tasks:

- Uses class CNielsenValidatorParameters to read and store command-line arguments
- Creates three objects and configures them based on the command-line parameter settings:
  - The *file reader* object (example source code provided)
  - The *logger / report-generator* object (example source code provided)
  - The *main processor* (an object of type CId3TagValidator).
- Invokes the file-reader Run() function, to process the entire input file.
- Calls the file-reader Report() function, to generate a summary report of the ID3 tag analysis.
- Releases all allocated resources, closes files, and exits.

NielsenValidatorParameters (.cpp and .h)

These files define the CNielsenValidatorParameters class, which reads command-line arguments, stores their settings, and makes those settings available to other modules for use in the configuration process. Its primary function is GetArgs(). All other public functions are get/set functions, used to access the private members of the class.

The options that are read from the command-line include:

- The type of input data to be processed/analyzed:
  - **–r** indicates that the input data to be processed is a stream of ID3 tags that are generated by any of these applications:
    - **PCM-to-ID3** The output of PCM-to-ID3 SDK sample application is a stream of encrypted Nielsen ID3 tags. These are complete tags: they include the ID3 tag header and the ID3 frame header, as well as the “www.nielsen.com” owner-id and the tag payload. The ID3 tags are arranged contiguously and in proper sequence in the stream.
    - **Tablet Sample Application (or CHLS log)**. The log produced by the tablet or browser sample application includes blocks of text, interspersed with encrypted ID3 tags. The Validator SDK buffers the incoming data, then parses the buffered data to find each “www.nielsen.com” string. It passes the encrypted tag (starting with “www,” but without the tag-header and frame-header) to various classes in the SDK libraries for further processing.
    - **Other Sources** – Actually, the SDK can process any file (text or binary) that includes encrypted tags that are not interrupted or truncated after the start of the “www” string.

  - **-t** indicates that the input data to be processed is a properly formed MPEG-2 transport stream that includes a Nielsen ID3-tag metadata elementary stream.
  - **-m** indicates that the input file is an m3u8 playlist. You may use this option if the m3u8 file provides an index to HLS (transport stream) segments that are stored on a local drive. It does not work when the m3u8 file uses a URL to reference the segments. This option is provided only to show that Validator SDK can process HLS segments, just as it processes an unsegmented transport stream.
- The input-type option (-r, -t, or -m) must be followed immediately by the full path name of the file to be processed.
- -**o,** followed by the full path name of the folder to hold the output report and the log file.
- **-p**, followed by an integer in the range 0 through 7, as defined on page 8 of this manual.
- **-s**, followed by the length of the periodic-report interval (in seconds, >= 10). This option is ignored if the –p option is set to 0. The recommended interval is 30 seconds.
- **-p**, followed by the program number of the program (in a multi-program transport stream) to process. If **–n** is not included on the command-line and, if the input is a multi-program transport stream, the application processes the first program listed in the program association table (PAT) of the transport stream.

The CNielsenValidatorParameters class uses the name of the input file as well as the path name of the output folder to generate the names of the output (log and report) files.

NielsenValidatorFileReader (.cpp and .h)

The **NielsenValidatorFileReader** files define the CNielsenValidatorFileReader class, whose main functions are:

- The **Constructor**, which accepts four arguments and opens the input file for reading. The arguments:
  - The name of the input file
  - A pointer to the logger object
  - A pointer to the main-processor object
  - An indicator of the file type (ID3-tag stream or transport-stream). Note that if you are processing HLS segments (-m), the file type is eTransportStreamDataType, just as if you were processing an unsegmented transport stream.
- **Run**() – Repeatedly reads buffers of data from file, passing each buffer to the main-processor object for processing and analysis.
- **Report**() – Calls the main-processor function, GenerateSummaryReport(). The logger object will “catch” the report data and store it in a file.

- **Destructor** – Closes the input file.

NielsenValidatorLogger (.cpp and .h)

The logger files define the CNielsenValidatorLogger class, a "callback class" derived from Id3TagValidatorLoggerBase. (An earlier section of this document provides details regarding callback classes). CNielsenValidatorLogger exposes these public methods:

- **Constructor** – Opens the output files. Requires these arguments:
  - nReportsToGenerate determines which of the 3 optional reports (periodic report, detailed tag listing, and comma-delimited tag listing) will be generated:
    - 0 – no optional reports
    - 1 – periodic report only
    - 2 – detailed tag listing only
    - 3 – periodic report, detailed tag listing
    - 4 – comma-delimited tag listing
    - 5 – periodic report, comma-delimited tag listing
    - 6 – detailed tag listing, comma-delimited tag listing
    - 7 – all three optional reports
  - Log file name (full path name of the log file to generate)
  - Tag file name (full path name of the file that will hold ID3 tags and/or NAES watermarks, if nReportsToGenerate is set to 2, 3, 6, or 7. Otherwise, set to NULL.
  - Report file name (full path name of the report file to generate). If nReportsToGenerate is set to 1, 3, 5, or 7, this report file will hold a series of periodic reports, followed by a single summary report.
  - Input file name (full path name of the input file to read, used only for reporting purposes)
  - Comma-delimited file name (full path name of the file that will hold a comma-delimited listing of each ID3 tag found in the content). If nReportsToGenerate is set to 4, 5, 6, or 7, this report file will hold the comma-delimited tag listing.
- **Destructor** – closes files, releases resources.
- `int LogMessage( int code, const char *message);`

This function is invoked by the SDK libraries whenever they have a status or error message to report. The sample application handles the log-message callback by storing in a log file each message and its corresponding code. However, your application may handle log messages differently. Simply modify **body** of the LogMessage function to handle the message and code as you see fit. Make certain, though, not to modify the function

declaration itself. It **must** match the name of the pure-virtual function defined in Id3TagValidatorLoggerBase.

> **Note** If the <code> argument is set to 11111 (cTagReportCode), the <message> is a listing of a single decrypted ID3 tag. The Nielsen Validator sample application stores tag listings in a separate file, named by the third constructor argument. If the <code> argument is set to 12121, the <message> is a comma-delimited report of the major fields of a single ID3 tag. The Nielsen Validator sample application stores the comma-delimited tag listing in a separate file, named by the last constructor argument. If the <code> argument is set to 22222 (cWatermarkReportCode), the message is a listing of a single audio code (also referred to as a watermark). The sample application stores watermark listings in the same file as detailed tag listings. The watermark listing begins with ****. By comparing the watermark time stamp to the time stamps of the EDUs in the next tag, you can determine whether or not the ID3 tags are properly synchronized with the audio watermarks.

- `int IssueStatusReport( ValidatorOutputFields_t *pValidatorStatus );`

The SDK libraries invoke this function whenever they have a report (periodic or summary) to deliver. The sample application sends these reports to a file; however, you may choose to handle the reports differently. Simply modify or replace the body of the IssueStatusReport() function to meet the needs of your application.

Note that the IssueStatusReport() function displays the pass/fail results by performing these steps:

1. In a for-loop, traverses the test array. For each test in the array:
   a. Assigns a label to the test
   b. Assigns a string to represent the test result (pass, fail, warning, or no status).
   c. Creates a string holding the count, min-occurrence, max-error settings for this test. Puts those three values between <> brackets. (For periodic reports, the sample application prints only the pass/fail status, but not the actual counts. In your output, you may elect to report as much or as little detail as you like).
   d. Writes the test line to file.

Also note that a PrintInfoTag() is provided to display the INFO tag in readable format.

Running the Sample Application

Please refer to the ID3 Tag Validator Application User's Guide for detailed instructions on running the Validator sample application and on interpreting the output files.