Grimm | AUDIO

LevelApi

Please read this manual before operating the software!

© 2012 Grimm Audio, All rights reserved.
Reproduction in whole or in part is prohibited.
Specifications subject to change without notice.

Manual written by Jorn T. Lemon

Software designed and developed by Jorn T. Lemon and Wouter D. Snel

Corresponding to software revision 261

Introduction

Thank you very much for selecting Grimm Audio's LevelAPI for your audio normalizing tasks. LevelAPI is build to support one of the most important achievements in the audio industry for decades: the change from peak normalization to loudness normalization. This "true audio revolution" started when the ITU submitted the BS.1770 'LKFS' (also known as LUFS) loudness metering standard in 2006. The European Broadcast Union EBU took the lead in building a broadcast recommendation upon this fundament, called R128. It was released in 2010. Eelco Grimm of Grimm Audio was one of the active members of the EBU PLOUD committee that created the recommendation. The committees work has been made possible by the aid of a piece of software developed by Grimm Audio's Wouter Snel and Jorn Lemon. Grimm Audio's LevelOne is the enhanced version of the actual software that made R128 happen and LevelAPI is the base of LevelOne's algorithms. We are very proud to offer you this great piece of software.

The libLevelAPI and the example are build on: Linux Version 2.6.32-5-686 (Debian 2.6.32-35) gcc v 4.3.5 (Debian 4.3.5-4). Linux Version CentOS i386 & x64

Usage

Implementation of the LevelAPI into you C++ environment is straight forward, especially if you have earlier experience with programming audio applications.

Dynamic library

The interface to the LevelAPI library are all C functions. A function call always makes use of a handle argument. Therefore a handle always needs to be created with LevelAPI_GetHandle() before usage of the library. And the handle always needs to be deleted with LevelAPI_DeleteHandle(handle) after the usage of the library is done. The handle allows you to create multiple LevelAPI instances at the same time and measure multiple sources at the same time, by using a new handle for each instance. The LevelAPIC.h header file contains all function definitions and convenience functions to load the dynamic library at run time.

Initialization

You will need to initialise the lib by calling LevelAPI_Init(handle, channels, samplerate, bitdepth) with each argument corresponding to the format of the audio data you will analyse and/or adjust.
Next you will need to setup the parameters in which the library will measure. This can be done with the functions Set and Get: BlockLength, RelativeGate, ChannelWeight and Calibration. But luckily there are 2 convenience functions that wrap the most common settings together in 1 call: LevelAPI_SetEBU_R128() and LevelAPI_SetATSC(). Which removes the need for the separate functions if you only need to normalize to the EBU or ATSC standard.
(These functions can be mixed by first calling ie. LevelAPI_SetEBU_R128() and then setting the Calibration manually with SetCalibration(handle, metertype, target)).

Analysing

You are now ready to analyse your audio sample data by calling LevelAPI_Analyse(handle, samples, numframes). Where 'samples' is a pointer to a interleaved floating point buffer and 'numframes' is the amount of sample frames. This means that a floating point array of 1200 'stereo' samples has 600 sample frames. and a array of 1200 '5.1 surround' samples has 200 frames (!) For best performance you should make as less calls to LevelAPI_Analyse() as possible. Which means that it is better to analyse big blocks of sample frames.

After you are done, or between analysing you can call LevelAPI_GetLevel(handle, metertype, channel) to get the meter levels in dB corresponding to the given arguments.

Dialog Gate

If the Dialog Gate is used (only specifically needed for ATSC measurements + Dialog intelligence) you will need to call the function LevelAPI_FinaliseDialog() before getting the final results for the types: Dialog_percentage and LUFS_Dialog. This is due to the delay that the gate causes, this function zero pads the end of the stream. See SetDialogGate() for more details.

Normalize/Process

After all samples are analysed you are ready to normalize your audio. If you use the same handle/instance of LevelAPI for which you used for analysis you can just call LevelAPI_Adjust(..) for you sample data to normalize it because the API already knows how much adjustment is needed. If you use a new handle/instance of LevelAPI you can set the adjust level manually with LevelAPI_SetAdjustLevel(..) and then call LevelAPI_Adjust(..).

Compression and Limiting

In cases where the dynamics of the audio does not allow one to normalize the audio to its target level, the compressor/limiter will kick in. This process also causes the audio to have a different loudness level. Which means a 2nd analysis and normalization step is needed.
The file processor of LevelAPI takes care of this extra step automatically. However the streaming part of LevelAPI will just use the compressor in the Adjust() function if deemed needed after analysis of the audio. You can check this with GetLevel() and 'CompressionAdjustment' as argument. If this is not 0 dB it means the complete stream will have to be processed again to get it exactly at the target level because the compressor has been used. Also note that the compressor/limiter never will be used on MPEG files due to the MPG compression only pure normalization/scaling is possible.

Example

The examples included in the package gives a simple example of the usage of the library.
In the windows example a 20 second 1000hz sine wave at -20dB is generated. It analyses the signal conforming to EBU R128 and prints the results afterwards to the terminal.

In the second part of the example the usage of the FileProcessor is also demonstrated which processes audio files of most common formats and lossless mpeg normalization.

LevelAPI library constants

These constants are used in the library function calls

enum Channels

```
{
    all     = -1,
    l       = 0,
    r       = 1,
    c       = 2,
    lfe     = 3,
    ls      = 4,
    rs      = 5,
    num_surround = 6,
}
```

These constants are needed because of the channel orders that differ between different surround standards.

enum PeakType

```
{
    Peak_dBSF,    // standard sample peak
    Peak_TrueLow, // corresponding to EBU standard
    Peak_TrueHigh // double precision (higher cpu load)
};
```

Used for the initialisation of the API.

enum SurroundOrders

```
{
    SMTPE_ITU_AC3,      // L-R-C-Lfe-Ls-Rs
    FilmDolbyDigital,   // L-C-R-Ls-Rs-Lfe
    DTS_ProControl      // L-R-Ls-Rs-C-Lfe
};
```

Used for the initialisation of the API, default is SMTPE_ITU_AC3.

enum MeterTypes

```
{
    LU,            // Calibrated LU level
    Max_M,         // Calibrated maximum momentary level
    Max_S,         // Calibrated maximum short term level
    Max_M_FS,      // Uncalibrated maximum momentary level (*)
    Max_S_FS,      // Uncalibrated maximum short term level (*)

    LUFS,          // Uncalibrated LU level (*)
    LUFS_UNGATED, // Uncalibrated LU level without gating (*)
    LUFS_Dialog,  // Uncalibrated LU level with dialog gating
    LUFS_Min,      // Minimum uncalibrated LU level (*)

    Dialog_percentage, // the percentage of ungated dialog content in the audio

    LRA,           // Loudness range
    LRA_Low,       // Uncalibrated minimum mark of the LRA,
                      can be used to draw the range on a meter
    LRA_High,      // Uncalibrated maximum mark of the LRA,
                      can be used to draw the range on a meter
    LRA_Min,       // Uncalibrated smallest value of the
                      LRA calculation (*)
    LRA_Max,       // Uncalibrated largest value of the
                      LRA calculation (*)


    Peak,          // the maximum sample peak (*)
    TruePeakLP,    // the maximum true sample peak (*)
    TruePeakHP,    // the maximum true sample peak,
                      double precision (*)
    AutoPeak,      // Automatically return the peak level that
                      has been selected in the API initialisation

    AdjustLevel,  // The adjust level
    AdjustLevel_NoPeak, // The adjust level without peak limiting

    LU_BS1771,      // Realtime Calibrated Momentary metering
    LU_BS1771Filtered, // LU_BS1771 with EBU recommended filtering for RT displays
    LU_BS1771_3s,   // Realtime Calibrated Shortterm metering
    LU_BS1771_10s,  // extended shortterm metering (*) available upon request
    LU_BS1771_30s,
    LU_BS1771_90s,
    LU_BS1771_270s,

    LUFS_BS1771,    // Realtime metering values without calibration LU -> LUFS
    LUFS_BS1771Filtered,
    LUFS_BS1771_3s,
    LUFS_BS1771_10s,
    LUFS_BS1771_30s,
    LUFS_BS1771_90s,
    LUFS_BS1771_270s,

    PPM,           // Peak Programme Meter
    PPM_Max,       // Max measured PPM

    PeakLoudness,           // PLR Peak loudness range
    CompressionAdjustment,  // if != 0. the compressor is needed
}
```

These types are mostly only used in the function LevelAPI_GetLevel(..).
Only 'LU' and 'AutoPeak' are also used in LevelAPI_SetAdjustTargetType(..)
(*) these values are generally not needed to display to the user, but can be used for testing purposes.

LevelAPI library functions

An explanation of all the functions and their arguments of this Dynamic library with C interface. include the 'LevelAPIC.h' in your source to call the following functions;

DLL handle

```
void *LevelAPI_LoadDLL(const char *libPath)
```

Loads and creates a handle to the dynamic library

```
void LevelAPI_UnloadDLL(void *dll_handle)
```

Unloads and deletes a handle to the dynamic library

LevelAPI handle

You need a handle to access all the functions. This handle will be the 1st argument of each function you call.

```
unsigned long LevelAPI_GetHandle()
```

Returns a valid handle or NULL if no success. A valid handle is necessary for all other function calls. (Do not manually create or change the value of this returned variable!)

```
void LevelAPI_DeleteHandle(const unsigned long)
```

When you are done with using the library you must call this function with each handle you created with GetHandle().

License Management

These license functions need to be used to set your license data. This data will not be saved automatically after you delete a handle. You can hardcode your license data in your software. Or use LevelAPI_SaveLicense() to save it to disk. Then, if the license data exists it will be used and these 'set' functions of the license data can be skipped.

```
bool    LevelAPI_License(const unsigned long)
```

Returns true if the current license is valid

- arg 1: Your LevelAPI handle

```
bool    LevelAPI_SetName(const unsigned long, const char*)
```

Set the license name (your webshop username or email adress)

- arg 1: Your LevelAPI handle
- arg 2: username, use an email address to ensure valid characters

---

```
bool    LevelAPI_SetSerial(const unsigned long, const char*)
```

Set the serial obtained from Grimm Audio

- arg 1: Your LevelAPI handle
- arg 2: serial string

```
bool    LevelAPI_SetResponse(const unsigned long, const char*)
```

Set the response obtained from Grimm Audio.

- arg 1: Your LevelAPI handle
- arg 2: response string

```
const char* LevelAPI_GetName(const unsigned long)
```

returns the name to which the license has been registered to.

- arg 1: Your LevelAPI handle

```
const char* LevelAPI_GetSerial(const unsigned long)
```

returns the serial to which the license has been registered to.

- arg 1: Your LevelAPI handle

```
const char* LevelAPI_GetChallenge(const unsigned long)
```

returns the challenge which will allow Grimm Audio to create a response key.

- arg 1: Your LevelAPI handle

```
const char* LevelAPI_GetResponse(const unsigned long)
```

returns the set response

- arg 1: Your LevelAPI handle

```
int         LevelAPI_GetTrialDays(const unsigned long)
```

returns the amount of trial days left in the currently set serial

- arg 1: Your LevelAPI handle

```
bool        LevelAPI_SaveLicense(const unsigned long)
```

Saves the license to Linux folder: /home/USER_NAME/LevelAPI
or Windows registery: HKEY_CURRENT_USER\Software\GrimmAudio\LVLA
It removes the further need for SetName() SetSerial() SetResponse on the current machine

- arg 1: Your LevelAPI handle

Main initialisation

```
bool    LevelAPI_Init(const unsigned long, const int, const unsigned int, const float)
```

The Init function needs to be called before using the library to Analyse or Adjust. But is not needed when used with just ProcessFile. (Because it will extract the arguments itself). This function and can be called later again to reset and prepare the API for a new measurement. returns 0 on success.

- arg 1: Your LevelAPI handle
- arg 2: Amount of channels of the source material, ie. 2 for stereo
- arg 3: The samplerate of the source material ie. 48000
- arg 4: The bit depth of the adjustment source material, which is used for dithering calculation when applying gain. ie. 32, 24 or 16.

```
int     LevelAPI_Revision()
```

Returns the current source version of the library.

```
bool    LevelAPI_SetEBU_R128(const unsigned long, const int)
bool    LevelAPI_SetATSC(const unsigned long, const int)
```

These are convenience functions to put the API in EBU R-128 mode or ATSC mode. returns 0 on success.
It combines the function calls to :
EBU R-128:

- SetRelativeGate(-10)
- SetBlockLength(100)
- SetCalibration(-23)
- SetPeakType(x)
- SetAdjustTargetType(LU) and
- SetAdjustTargetLevel(0.)

ATSC:

- SetRelativeGate(0)
- SetBlockLength(100)
- SetCalibration(-24)
- SetPeakType(x)
- SetAdjustTargetType(LU) and
- SetAdjustTargetLevel(0.)

For more explanation of these functions go to the 'Advanced Settings' below.

- arg 1: Your LevelAPI handle
- arg 2: a peak type from the 'PeakTypes' enum list

```
bool    LevelAPI_SetDestructive(const unsigned long, const bool destructive)
```

This function sets the treatment of the analysis samples.
returns 0 on success.

- arg 1: Your LevelAPI handle
- arg 2: Default is true. If set to false the sample data entered into the Analysis function will not be changed. Depending on your implementation this may be desirable.

```
bool    LevelAPI_SetAdjustLevel(const unsigned long, const float)
```

The LevelAPI_SetAdjustLevel function can be used to optionally set the API Adjust Level manually. This function is not needed if LevelAPI_Analyse (with the same handle) is used to obtain the AdjustLevel.
returns 0 on success.
This allows you to first Analyse the audio and save the AdjustLevel. And normalise it later by using LevelAPI_Adjust(..) after calling LevelAPI_SetAdjustLevel(..) with:

- arg 1: Your LevelAPI handle
- arg 2: The earlier saved AdjustLevel in dB

Processing

```
bool LevelAPI_Analyse(const unsigned long, float *, const unsigned int, const int)
```

The main function that analyses the sample data.
returns 0 on success.

- arg 1: Your LevelAPI handle
- arg 2: A pointer to the (interleaved) sample array, these samples will be altered for the analysis process. if you want the array of samples to remain unchanged; then call the function 'LevelAPI_SetDestructive(false)'
- arg 3: The amount of sample frames in the array, for best performance it should be 256 or bigger. important: this value is in sample frames, in other words (samples/channels) ie. 512 samples in a interleaved stereo buffer contains 256 sample frames
- arg 4: **optional argument** default is off, tc = -1. Is only needed in select production environments (editing/mixing) with real time metering. In these environments an audio on a timeline is being updated with new audio. Keeping track of the timestamps allows LevelAPI to quickly recalulate the loundness of the entire timeline when just a small piece is updated. tc is an integer represening the timeline position of the incoming audio in 0.1 sec units. (ie. tc = 357, means 35.7 seconds). This value only has to be given at the start of the new timeline position. Any following calls to Analyse() will keep track of the tc position.
  Please note: This functionality will require more memory when used, since it will keep track of the entire timeline data. Therefore do not use it with semi-inifinite audio streams(!)

```
bool    LevelAPI_Adjust(const unsigned long, float *samples, const unsigned int numfram
es)
```

Normalises the sample data to the AdjustLevel that has been calculated by using LevelAPI_Analyse(..) or has been set by LevelAPI_SetAdjustLevel(..)
returns 0 on success.

- arg 1: Your LevelAPI handle
- arg 2: A pointer to the sample array
- arg 3: The amount of sample frames in the array. important: this value is in sampleframes, in other words (samples/channels) ie. 512 samples in a interleaved stereo buffer contains 256 sampleframes

---

```
bool    LevelAPI_ProcessFile(const unsigned long, const char *inFilePath, const char *o
utFilePath)
```

Opens the input file and analyses it with the current settings of the library. The Init() call is not needed (this data will be extracted form the file itself). Returns 0 on success.
This function is not available yet in the Windows 64 bit version of this library.

- arg 1: Your LevelAPI handle
- arg 2: The complete (non-relative) path to the input file
- arg 3: The complete (non-relative) path to a non-existing output file. If this path is NULL the input file will only be analysed.

Realtime helper functions

```
bool    LevelAPI_HasTimelineGap(const unsigned long)
```

When the 4th argument of Analyse() is used to track and update a timeline. You can use this function to detect if there are gaps in the timeline data. ie. there is a middle part missing. Then this function will return true.

- arg 1: Your LevelAPI handle
- arg 2: The requested handle from LevelAPI_GetHandle

```
bool    LevelAPI_UpdateValues(const unsigned long, const int tc)
```

When the 4th argument of Analyse() is used to track and update a timeline. You can use this function to scrub through the timeline data. ie. when the used moves the timeline position the realtime loudness data will be updated to the given position.

- arg 1: Your LevelAPI handle
- arg 2: tc is an integer represening the timeline position of the incoming audio in 0.1 sec units. (ie. tc = 357, means 35.7 seconds)

Results

```
bool    LevelAPI_FinaliseDialog(const unsigned long)
```

Only when the dialog gate is used this function needs to be called before calling the final results of the audio stream. Only do this once per stream! This function exists because the dialog gate has a ~2 sec delay, therefore the end of the stream needs to be zero padded to get the final results. Which means calling this function before analysis is finished will corrupt the results. Returns 0 on success.

- arg 1: Your LevelAPI handle

```
float   LevelAPI_GetLevel(const unsigned long, const unsigned int, const int)
```

Request any of the meter levels from the 'MeterTypes'

- arg 1: Your LevelAPI handle
- arg 2: The meter type from the 'MeterTypes' enum list
- arg 3: The channel selection from the 'Channels' enum list, default is 'all' specific channels are only applicable for Peak

---

```
float   LevelAPI_GetAdjustLevel(const unsigned long, const bool useNoPeak = false)
```

A convenience function to get the calculated adjust level to normalize the audio in dB. This function can be used to just obtain the needed adjustment level, and use this to manually adjust your audio data to that level.

- arg 1: Your LevelAPI handle
- arg 2: Default is false. Use true if you need the adjust value without correction of any clipping after the adjustment.

```
bool    LevelAPI_IsAdjustLevelLowered(const unsigned long)
```

IsAdjustLevelLowered can be used to check if the (true)peak level did not allow a full normalize adjustment. ie. it returns 'true' if the peak was too high to do a full adjustment without the risk of clipping the signal The user should be warned if this is true because the adjustment will result in a value lower than the calibrated LU target

Advanced setings

These functions below are generally not needed, they alter the way the measurements are done.
For EBU R128 or ATSC compliance please use the LevelAPI_SetEBU_R128 and LevelAPI_SetATSC functions
The 'set' functions return 0 on success.

```
bool    GetDialogGate(const unsigned long)
bool    SetDialogGate(const unsigned long, const bool)
```

- arg 1: Your LevelAPI handle
- arg 2: boolean; default is off / false

```
int     LevelAPI_GetSurroundOrder(const unsigned long)
bool    LevelAPI_SetSurroundOrder(const unsigned long, const unsigned int)
```

For surround sources the order of the surround channels in the interleaved audio data can be set here by using the types from the 'SurroundOrders' enum list

- arg 1: Your LevelAPI handle
- arg 2: One of the types from the 'SurroundOrders' enum list

---

```
float   LevelAPI_GetChannelWeight(const unsigned long, const int)
bool    LevelAPI_SetChannelWeight(const unsigned long, const int, const float, bool con
st translate)
```

Get and Set the channel weight used to measure for each channel, which is only relevant for surround sources. You may use the LevelAPI_GetSurroundOrder() function to do this automatically.

- arg 1: Your LevelAPI handle
- arg 2: A channel from the Channels list
- arg 3: The channel weight in dB
- arg 4: translate the channel order to your selected SurroundOrder (default=false)

```
bool    LevelAPI_SetTaskIsComponent(const unsigned long, const bool is)
```

Not applicable. future use.

```
bool    LevelAPI_SetLUType(const unsigned long, const int)
bool    LevelAPI_GetLUType(const unsigned long)
```

Use the calibrated LU or non-calibrated LUFS value for the adjustment. Depreciated, only needed in very rare cases.

- arg 1: Your LevelAPI handle
- arg 2: The requested handle from LevelAPI_GetHandle.
- arg 3: the 'LU' or 'LUFS' meter type from the 'MeterTypes' enum list

---

```
int     LevelAPI_GetAdjustTargetType(const unsigned long)
bool    LevelAPI_SetAdjustTargetType(const unsigned long, const int)
```

Request the current target meter type used for the AdjustLevel, corresponing to the 'MeterTypes' enum list (always LU for R128 and ATSC). Or set the current target meter type corresponing to the 'MeterTypes' enum list, valid values are LU and AutoPeak.

- arg 1: Your LevelAPI handle
- arg 2: A meter type from the 'MeterTypes' enum list

```
float   LevelAPI_GetAdjustTargetLevel(const unsigned long)
bool    LevelAPI_SetAdjustTargetLevel(const unsigned long, const float)
```

Get and Set the target level, for EBU and ATSC compliance this should be 0 (default)

- arg 1: Your LevelAPI handle
- arg 2: The taget level in dB

```
int     LevelAPI_GetPeakType(const unsigned long)
bool    LevelAPI_SetPeakType(const unsigned long, const int)
```

Get and Set the peaktype used to measure.

- arg 1: Your LevelAPI handle
- arg 2: A peaktype from the PeakTypes list, which is also used for LevelAPI_IsAdjustLevelLowered()

```
int     LevelAPI_GetBlockLength(const unsigned long)
bool    LevelAPI_SetBlockLength(const unsigned long, const int)
```

Get and Set the block length used to measure.

- arg 1: Your LevelAPI handle
- arg 2: The length in miliseconds, default is 100.

```
float   LevelAPI_GetRelativeGate(const unsigned long)
bool    LevelAPI_SetRelativeGate(const unsigned long, const float)
```

Get and Set the relative gate used to measure.

- arg 1: Your LevelAPI handle
- arg 2: The gate in dB, default is -10. for EBU and 0. for ATSC

```
float   LevelAPI_GetCalibration(const unsigned long, const int)
bool    LevelAPI_SetCalibration(const unsigned long, const int, const float)
```

Get and Set the calibration used to measure.

- arg 1: Your LevelAPI handle
- arg 2: Meter type LU or Peak from Channels list
- arg 3: The calibration in dB. default is 0.

```
int*    LevelAPI_GetLUFSTable(const unsigned long, int &length)
```

Not applicable. future use.

```
std_vector<float> LevelAPI_ExportVector(const unsigned long)
```

Not applicable. future use.

Technical specifications

EBU setting uses the official EBU R128 settings:

- ITU BS.1770-2 measurement, including -10 LU 'background sound' gate
- 0 LU = -23 LUFS
- Adjustment of the target level to 0 LU +/- 6 LU

ATSC setting uses the 2009 version of the ATSC A/85 recommendation:

- ITU BS.1770-1 measurement, without relative gate
- 0 LU = -24 LKFS
- Adjustment of the target level to 0 LU +/- 6 LU

Surround channel order

The LUFS measurement has a different weighting for the surround speakers. Therefore LevelAPI must know the channel order of multichannel files in your facility. There are three options:

- SMPTE/ITU (L-R-C-Lfe-Ls-Rs)
- Film (L-C-R-Ls-Rs-Lfe)
- DTS (L-R-Ls-Rs-C-Lfe)

References

http://tech.ebu.ch/loudness provides all kinds of information about the EBU R128 broadcast loudness recommendation. The official R128 documents and guidelines can be found, as well as introduction papers and videos.

At http://www.atsc.org/cms/index.php/standards/recommended-practices/185-a85-techniques-for-establishing-and-maintaining-audio-loudness-for-digital-television the ATSC A/85 recommendation is available for download.

3rd party libraries

mpg123

http://www.mpg123.de/
free software licensed under LGPL 2.1

libsndfile

http://www.mega-nerd.com/libsndfile/#Licensing
free software licensed under LGPL 2.1

http://www.gnu.org/copyleft/lesser.html

Grimm Audio CV
Strijpsestraat 94
5616 GS Eindhoven
The Netherlands

Phone: +31 (0)40-213 1562

Email: info@grimmaudio.com
Website: http://www.grimmaudio.com