



## Open.tl Parser Demonstration

Using Bison to create an Open.tl parser

Version Beta 1.2a - 2000-01-10

### Geek Start

- Unzip OpenTL7.zip. It contains subdirectories. It may be convenient to place it at C:\, as some paths in the projects are hard-coded.
- Complete MSVC 4.2 projects are provided for :
  - \* TLTest (C:\OpenTL\parser\ tltest.mdp) (the demo itself)
  - \* Bison ((C:\OpenTL\Bison\bison\ VCBison.mdp)
  - \* Flex (C:\OpenTL\Bison\Flex\ VCFlex.mdp)
- TLTest is a console app which takes an OpenTL EDL file as input and outputs to console. TLTest.cpp contains 'main' and includes MyReader.h which contains the demonstration 'MyGet..' methods.
- If you have unzipped the distribution to C:\OpenTL\ you need only build TLTest. In this case pre-built versions of Bison.exe and Flex.exe are provided. If you have stored the OpenTL directory on another drive you will need to edit the files.h of the Bison project to replace the hard-coded paths to bison.sim and bison.hai and re-build Bison and Flex.
- Write your EDL interface code into TL\_EDL.BIS provided with the TLTest project. Callbacks to "MyReader" demo methods have been provided for all OpenTL data types in this file.
- Bison.exe and Flex.exe must be run to generate MrTLPar.cpp and MrTLFlx.cpp. This is done automatically as a 'Custom Build' in the TLTest project.
- MrTLPar.cpp and MrTLFlx.cpp (and related headers) are compiled and linked to TLTest.

***End of "Geek Start" – If this concise explanation is sufficient, no need to read further***

### Introduction

The Open.tl demo parser uses a parser generator called Bison to create parser code modules, which in turn are integrated into your application. The generated parser code provides a basic framework which can parse Open.tl data to your application's internal data types.

There is no requirement for you to use this approach but it is highly recommended for two reasons. First, the technology is an excellent method for parsing ASCII files in general and because Open.tl is written specifically to exploit it. Second, the demonstration shows exactly how the Timeline MX2424 and MMR8 products parse Open.tl files, so applications developed with this same coding approach should exhibit good interoperability with Timeline products.

Be certain you can legally use Bison in your application. Please review *Conditions for Using Bison* at [http://www.gnu.org/manual/bison-1.25/html\\_chapter/bison\\_2.html#SEC2](http://www.gnu.org/manual/bison-1.25/html_chapter/bison_2.html#SEC2) and the *GNU GENERAL PUBLIC LICENSE* at [http://www.gnu.org/manual/bison-1.25/html\\_chapter/bison\\_3.html#SEC3](http://www.gnu.org/manual/bison-1.25/html_chapter/bison_3.html#SEC3). Further information is available from Timeline marketing - contact Ron Franklin at [ronf@timelinevista.com](mailto:ronf@timelinevista.com).

## A Brief History of Bison

Bison is a member of a family of parser generators originally developed for compiler technologies, in particular the compiler pre-processor. Here, the programming language is defined in strict BNF (Backus Naur Form), LALR(1) syntax. The parser generator is used to create a parser for this language. This parser "token-izes" the elements of the language which, in turn, are used to interpret the meaning of the source code. C and C++ are such languages.

These technologies trace their roots to early programming environments, in particular ALGOL 60 (around 1960). Most modern computer languages have their roots in this. The BNF specification itself originated here (depending on how you interpret the lore). (see Reference 4). Most modern parser generators, like Yacc ("Yet Another Compiler Compiler") are derived directly from this lineage. (see Reference 3). There are many parser generators and related tools which have been developed for special purposes and environments (see Reference 5).

Bison is a direct relative of Yacc (a 'bison' is related to a 'yak'!) (see REFERENCES in the Bison distribution). The Bison manual is at Reference 1. The text of this manual is also included in the Bison distribution (Bison.info-n). See Reference 2 for a detailed explanation of Bison.

## OPEN.TL

While these technologies were developed for compilers, the principles are applicable to any data structure. HTML and XML, for instance, are related to this train of thought - they can be parsed with code generated by one of these parser generators. Similarly, Open.tl is written explicitly for parsing with code generated by Bison. Yacc itself, or some other parser generator, might be used to parse Open.tl, but Bison has been selected as most appropriate because it is robust, transportable, and relatively simple.

The Open.tl specification is documented in OpenTLx.DOC, included with this distribution.

## Bison References

1. The Bison Manual:  
*Bison - The YACC-compatible Parser Generator November 1995, Bison Version 1.25*  
[http://www.gnu.org/manual/bison-1.25/html\\_chapter/bison\\_toc.html#TOC7](http://www.gnu.org/manual/bison-1.25/html_chapter/bison_toc.html#TOC7)
2. Bison principles:  
[http://www.gnu.org/manual/bison-1.25/html\\_chapter/bison\\_4.html](http://www.gnu.org/manual/bison-1.25/html_chapter/bison_4.html)
3. Yacc and Lex:  
[http://members.xoom.com/\\_XOOM/thomasn/y\\_man.pdf](http://members.xoom.com/_XOOM/thomasn/y_man.pdf)
4. BNF:  
<http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html>
5. Other Parser Generators and Tools:  
<http://www.idiom.com/free-compilers/CATEGORY/compiler-1.html>

## Alrighty, then. How do I do it?

The Open.tl edl format specifies an ASCII file with BNF LALR(1) syntax. The Bison parser generator and the Flex lexical analyzer generator are used to create .cpp source files which understand the rules of Open.tl. These are then included in the application.

Bison works in conjunction with Flex. The yyparse function created by Bison (the parser generator) relies on tokens handed to it by the yylex function which is created by Flex ('fast lexical analyzer generator'). This function is sometimes referred to as a lexical scanner.

For more detail on how Bison uses Flex, or more specifically, how Bison uses the yylex function created by Flex, see "The Lexical Analyzer Function yylex" in the Bison documentation. Other lexical scanners might be used with Bison, or you could roll your own, but these two, Bison and Flex, are typically partners in crime. The Open.tl demonstration uses them together.

## Using Bison and Flex

Bison.exe and Flex.exe are pre-built and reside in the C:\OpenTL\Bison\BF\_EXE\Debug and \Release directories. You can use them as they stand on Win95,98, or NT if you have placed the OpenTL.

Since Bison and Flex create generic c source files, the platform bison and Flex are run on does not matter. If you are using some other development environment or are targeting non-windows platforms you could still use these Windows versions of Bison and Flex running on a Windows machine to create the source files.

If you need or want to build them, two MSVC 4.2 projects reside in C:\OpenTL\Bison\ subdirectories - \Bison\VCBison.mdp and \Flex\VCFlex.mdp. Building these projects creates Debug and Release versions of each, and 'Custom Build' scripts in the projects copy them to BF\_EXE\Debug and BF\_EXE\Release as Bison.exe and Flex.exe. These projects include the required Bison and Flex source files which Timeline has modified for Win32 compatibility.

*Important:* There is one reason you may need to re-build Bison. The file C:\OpenTL\Bison\Bison\files.h contains a hard-coded path to the location of Bison.sim and Bison.hai, which Bison needs when it runs. These files are the special gnu code which is patched into the MrTLPar.cpp source file by Bison when it runs. If you install Open.tl directory somewhere other than C:\ you will need to change this line and rebuild Bison. The code you need to change is commented in files.h - /\* TIMELINE (BEH) Hardcore paths to Bison.simple and Bison.hairy \*/. As distributed, these files are in the BF\_EXE\Debug and \Release directories. (They are copies of Bison.simple and Bison.hairy)

If you need to build for another environment, the complete unmodified gnu distributions are included in C:\OpenTL\Bison\ as bison-1\_25\_tar.gz and flex-2\_5\_4a\_tar.gz. These distributions were obtained from gnu.org at <http://www.gnu.org/order/ftp.html> >> <ftp://ftp.gnu.org/gnu/bison/bison-1.25.tar.gz> (288KB) and [ftp://ftp.gnu.org/gnu/flex/flex-2\\_5\\_4a\\_tar.gz](ftp://ftp.gnu.org/gnu/flex/flex-2_5_4a_tar.gz).

## TLTest - the OpenTL Demonstration

C:\OpenTL\parser\ contains TLTest.mdp, an MSVC 4.2 project which builds the demonstration program. The resulting TLTest.exe is a console app which takes an Open.tl file as input and does printf's of the Bison operations in progress and the parsed values to console (redirect it to a file, if you want). In addition, a command-line .mak file, TLPARS.MAK, is included for those who prefer this type of environment.

C:\OpenTL\parser\ and the project include to files especially important to the Bison and Flex operations, TL\_EDL.FLX and TL\_EDL.BIS.

The TL\_EDL.FLX file describes the Open.tl token names and return values to Flex. Timeline has written this file for you, and it includes some utility routines for returning simple values. There should be no reason for you to modify this file.

The TL\_EDL.BIS file describes the 'tokens' and 'rules' of Open.tl for Bison. Timeline has written the body of this file for you. **It is here you will find the most interesting demonstration function calls and where you must modify or write your own routines for extracting information from the parser.**

When Bison and Flex are run on TL\_EDL.BIS and TL\_EDL.FLX, Bison creates the parser source module (MrTLPar.cpp) and a header file (MrTLPar.h), and Flex creates the scanner source module (MrTLFlx.cpp).

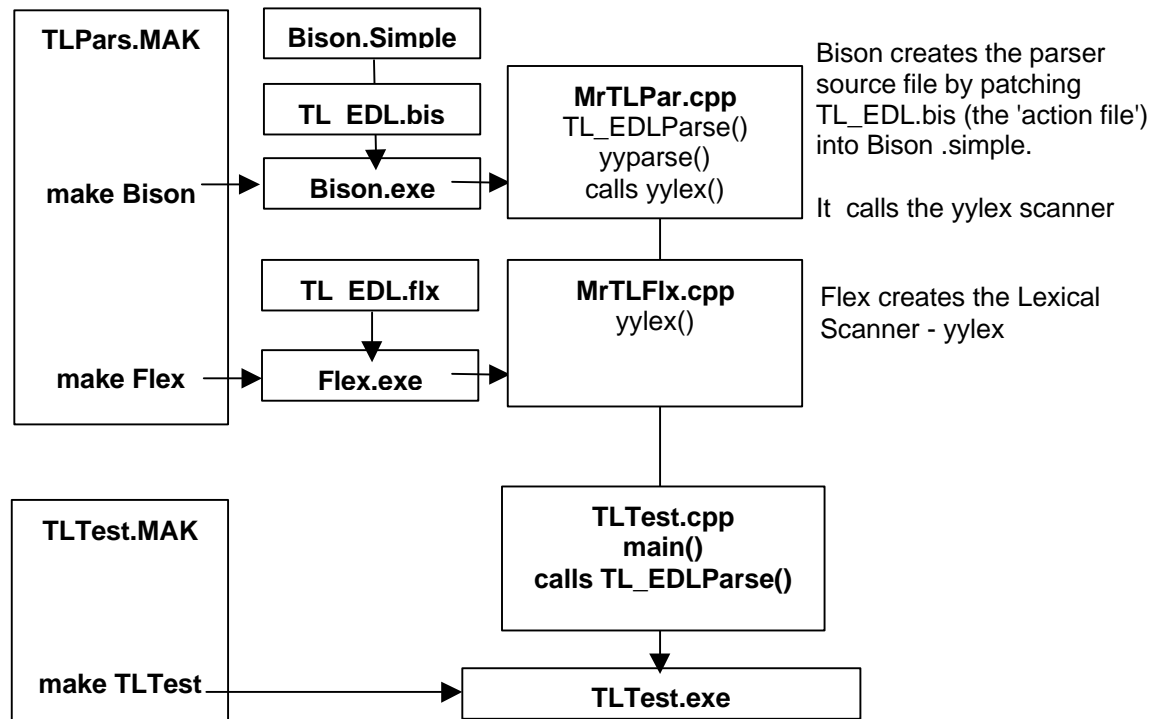
(For details on how Bison accomplishes all this please read the Bison documentation. If you are unfamiliar with Bison the effort is worthwhile not only for Open.tl purposes but also because the Bison technology is a very valuable tool in its own right. We think the knowledge is worth the investment.)

The TLTest project performs these steps automatically using 'Custom Build' scripts attached to the to TL\_EDL.BIS and TL\_EDL.FLX entries in 'Project Settings'. (If you are compiling from the command line, TLPARS.MAK actually makes two calls to nmake, one for Bison, and one for Flex.)

(Note: Bison outputs a .c file, MrTLPar.c. The 'Custom Build' scripts (or .mak) copy this file to a .cpp name, MrTLPar.cpp. It is the .cpp file that is used.)

MrTLPar.cpp and MrTLFlx.cpp are then combined with an application's modules to create a working program. In the case of the demonstration program, 'main' of TLTest.cpp calls the parser with a YYPARSE\_PARAM pointer input parameter (described below) which points to a c++ class called MyReader, declared in MyReader.H. MyReader contains data members and "MySet" methods. These "MySet" methods are called-back by the parser to communicate data types and values to MyReader. The instructions for this callback are written in TL\_EDL.BIS.

All these modules (MrTLPar.cpp, and MrTLPar.h as created by Bison, MrTLFlx.cpp as created by Flex, MyReader.H, and TLTest.cpp) are compiled and linked to create TLTest.exe by the project (or TLTest.mak). Running the resulting TLTest.exe with a valid Open.tl EDL example as input will fill the MyReader data members with data extracted from the sample file. As it does, printf's of each value are sent to console, with printf processing messages from TL\_EDL.BIS interspersed.



Headers and dependencies not shown

The trick to this callback mechanism is the YYPARSE\_PARAM. Bison provides for passing anything into the parser because YYPARSE\_PARAM is a void\* pointer. By casting this input pointer back to its correct form when referring to it in TL\_EDL.BIS you can call anything you might want. For details on this see "Calling Conventions for Pure Parsers" in the Bison doc.

To make this happen in the demonstration, several things are required. First, c++ class MyReader is declared in MyReader.h. It includes data members intended to collect data from the parser and the MySet callback methods intended to be called by the parser. You can extend this as you feel fit for your application.

Next, #include MyReader.h in both TLTest.cpp and TL\_EDL.BIS.

In the 'utility defines' section of TL\_EDL.BIS, add some defines so the void\* YYPARSE\_PARAM is cast to our MyReader type:

```

.....
/* utility defines */

// TL DEMO ////////////////////////////////////////
// define YYPARSE_PARAM to something

#define YYPARSE_PARAM myRdr

// and cast it to the type you are using.
// Methods in class MyReader can now be called

#define TLRdr ((MyReader*) myRdr)
.....

```

Now, TLRdr is of type MyReader, so we can call the MySet methods as shown in the 'project\_name:' rule in TL\_EDL.BIS:

```
.....
    project_name:      PJNM string_val
                      {
//                                printf("Parsing project file %s\n", $2);
// TL DEMO //////////////////////////////////////
// call MyReader::MySetProjectName
// input is cast to (char*), see string_val: rule
                                TLRdr->MySetProjectName((char*) $2);
                      }
                      ;
.....
```

Finally, in 'main' of TLTest.cpp, instantiate MyReader and pass it as input to a call to the parser:

```
.....
    MyReader myreader; // instance of class MyReader
    .....
    // call the parser with myreader as the YYPARSE_PARAM
        TL_EDLparse(&myreader);
    .....
.....
```

So, when the scanner returns the PJNM token to the parser, TLRdr->MySetProjectName((char\*) \$2); is called, passing the string up to MyReader.

That's the basic idea.

## Open.tl Application Design Considerations

The following are comments about the Open.tl specification (OpenTL.DOC) you may find helpful.

### Project and Track Files (Section 1.0)

Open.tl is made up of a set of files, the Project file and some number of Track files. So the first mission is to open the project file and find out what else you need to do. You can first collect the project data (title, sample rate, etc) and then get a list of the Track files, preparing to open each and read the event information of each track.

### Properties and Text Conventions (Section 2.0)

This is the Open.tl grammar, essentially the BNF which describes the language. This is mostly the business of Bison and Flex. The rest of the document describes each of the elements of Open.tl. These are the token names and values.

### FILE or DataSet Reference (Section 3.0)

This is the definition of the information used to reference files, both Open.tl EDL files, and audio media. The three properties tell you the type of file its and where it is. Read the Appendix section 6.1 Volume Names carefully. It is important this information is correct so other Open.tl compliant applications can find the media files.

## Project Files (Section 4.0)

The project file contains the project's general information and a list of Track files. As mentioned, this is where you collect a list of the Track files and prepare to read them.

Of particular interest is the Sample Rate (4.3). The SMRT (Sample Rate) values are the supported sample rates. Be sure you choose the right one!

Also, note that the "Frame Rate" (FRRT) should be used only for display purposes, such as converting a sample position to SMPTE timecode.

## Track Files (Section 5.0)

Each Track file represents one output audio track along the timeline.

TrackName (TKNM) holds a string with a user assigned track name. This should be the same as TrackName in the TRAK of the TrackList (TKLS) in the Project File. You should not rely on this to relate Track Files to the Project tracks - use the FILE or Dataset Reference instead.

StartTime (STTM) is an absolute start time of the track in samples. For instance, a StartTime of 48000 in 48K sampling would indicate an absolute StartTime of 00:00:01:00 if converted to timecode (this conversion may be approximate depending on the framerate). Thus, each track in a project is anchored to an absolute position along the timeline.

Each track may begin at different times. Thus the beginning of the project's timeline is the earliest (lowest) StartTime of any of the project's tracks.

The end of the project's timeline is the last sample of the last event in any track. The last sample in any track is the track's last event's EditPoint plus its 'Length-to-play', which is (EditLen - RampUpOffset - RampDownOffset + RampUpLen + RampDownLen) (see Clips and Transitions).

The edit list (EDLS) contains the events. The track may also contain an optional Gain List (GNLS) and/or optional Mute List (MULS)

## Directory (Folder) Naming Conventions (Appendix 6.2)

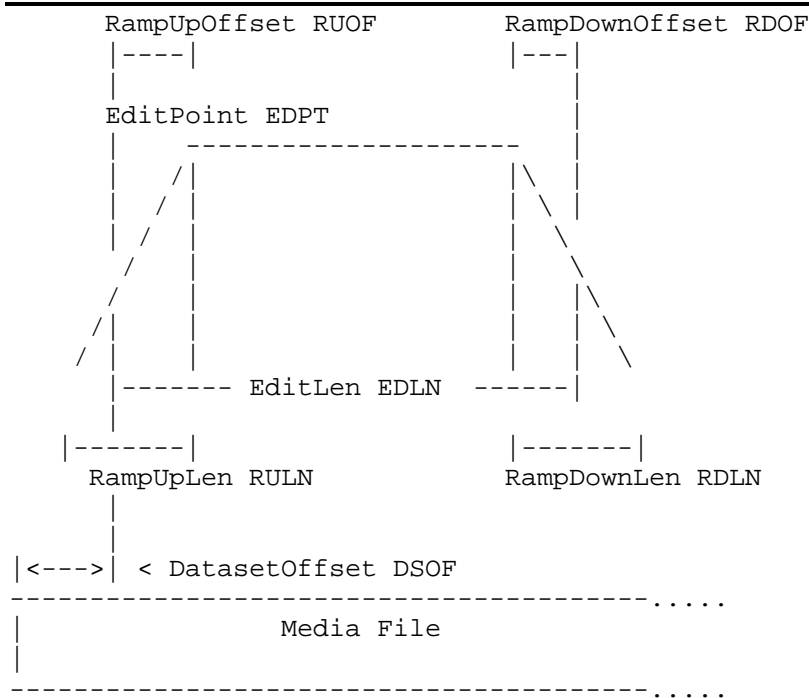
By convention, the following directory (folder) names will be used for OpenTI projects:

```
<MyProject>                // Any reasonable project name
  MyOpen.TL
  <Track Files>              // By convention "Track Files"
    MyTrack1                 // Track files....
    MyTrack2
  <Audio Files>              // By convention "Audio Files"
    Audio1                   // Audio files....
    Audio2
```

## Clips and Transitions (Section 5.3)

It is the application developer's responsibility to map Open.tl's edit representation to the application's required internal data representation, and back again.

In Open.tl, each event represents a "clip". Transitions (crossfades) are represented as ramp up and down information attached to clips - there are no explicit transition types.



RampUpOffset must be  $\geq$  zero  
RampUpLen must be  $\geq$  zero  
RampDownOffset must be  $\geq$  zero  
RampDownLen must be  $\geq$  zero

RampUpOffset must be  $\leq$  RampUpLen  
RampDownOffset must be  $\leq$  RampDownLen

First sample to play = DatasetOffset + RampUpOffset - RampUpLen

Length to play = EditLen - RampUpOffset - RampDownOffset + RampUpLen + RampDownLen

Clip EditPoints must be in ascending order (No overlapping splice points)

Zero Length clips are NOT allowed

---

Each event references an audio media file. The Event FILE reference (EVDS) contains a FILE or DataSet Reference (Paragraph 3.0). The FILE simple properties will tell where the file is and SNTY gives kind of media.



EditPoint (EDPT) is an absolute sample position along the output track's timeline. For example, if the track's start time (STTM) is 172800000 and the EditPoint is 172910000, there are 110000 samples between the beginning of the track and the EditPoint ( $172910000 - 172800000 = 110000$ ).

DataSetOffset (DSOF) is the number of samples from the beginning of the media file (zero is the first sample in the media file) to the EditPoint. It tells you the sample in the media file which corresponds to the EditPoint.

The EditPoint 'anchors' the event to the timeline, and all the event offset and length values are relative to it (in samples). Calculating any of these points is a simple matter of adding and subtracting the lengths and offsets relative to this absolute timeline position.

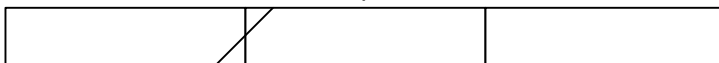
EditPoint and EditPoint plus EditLen (EDLN) define the two end points of the 'event'. However, playback of the referenced source *can occur both before and after* these two 'end points', depending on RampUpOffset (RUOF), RampUpLen (RULN), RampDownOffset (RDOF) and RampDownLen (RDLN). In other words, these end points actually define the boundary between Open.tl's representation of events, not the beginning and ending of the actual playback of the clip.

To illustrate, consider three cuts:



Here the values of RampUpOffset, RampUpLen, RampDownOffset and RampDownLen are all zero on all the events - playback is coincident with the event boundaries.

Now consider when the first clip cuts out, and the second fades in:



Here the values of RampUpOffset and RampUpLen of the second event describe playback beginning before the EditPoint at zero gain, and ramping up to unity. Playback begins at EditPoint plus RampOffset minus RampUpLen, and reaches unity RampUpLen after this point.

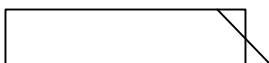
Note the event end points are still butted up - the end points (EditPoint and EditPoint plus EditLen) define the boundaries of the event 'bricks' laid end to end. This is always true.

The playback points can vary around these boundaries on both the outgoing and incoming events. Consider a classic symmetrical crossfade:

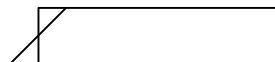


Here, the first event's RampDown Offset and RampDownLen, and the second event's RampUpOffset and RampUpLen describe the crossfade.

The first event looks like:



And the second event looks like:



Both are playing back on the same output track, of course.

Open.tl events can describe an asymmetrical crossfade:



It can also fade in and out on a single event:



There are no 'silence' or 'fill' events in Open.tl. The position of the event along the output timeline is explicit - if there is no event preceding it, nothing is playing back. So a fade up can look like:



Here, nothing precedes the event, so nothing is playing back before the event - it is ramping up from nothing, and nothing is very similar to silence.

Any combination of valid values is possible, so Open.tl can describe a very wide range of event types and transitions.

## Slider Value (Fader Level) to Db and Gain Conversion (Section 5.4)

In Open.tl, volumes are represented as "slider value" or "fader level". The following shows the exact relationship of this to dB and Amplitude (or linear gain)

Slider Value		dB	Amplitude (linear gain)
1.0	-----	+6 Db	2 X gain
	-----	0 Db	Unity gain
	-----		
	-----		
	-----		
0.0	-----	-∞ Db	0 gain

$$\text{dB} = 40 * \log(10^{6/40} * S) \quad \text{"dB} = 40 \text{ times log of (10 to the 6/40th times S)"}$$

$$\text{Amplitude} = (10^{6/40} * S)^2 \quad \text{"Amp} = (10 \text{ to the 6/40th times S) squared"}$$

where S is "Slider Value"

## Gain List

Each track may have an optional Gain List (GNLS). This represents a control channel running parallel to the track's timeline. The GainEvents (GNEV) contain a GainPoint (GNPT) which is the absolute timeline position in samples, and a GainLevel (GNLV) which is a SliderValue as described in *Slider Value (Fader Level) to dB and Gain Conversion*

## Mute List

Each track may have an optional Mute List (MULS). This represents a control channel running parallel to the track's timeline. The MuteEvents (MUEV) contain a MutePoint (MUPT) which is the absolute timeline position in samples, and a MuteLevel (MULV) which has a value of either 1 (Mute is ON = silence) or 0 (Mute is OFF = play normally).