DeliAPI 2

Documentation Version 1.0

The player plugin API for DeliPlayer Version 2.50 and higher

Introduction

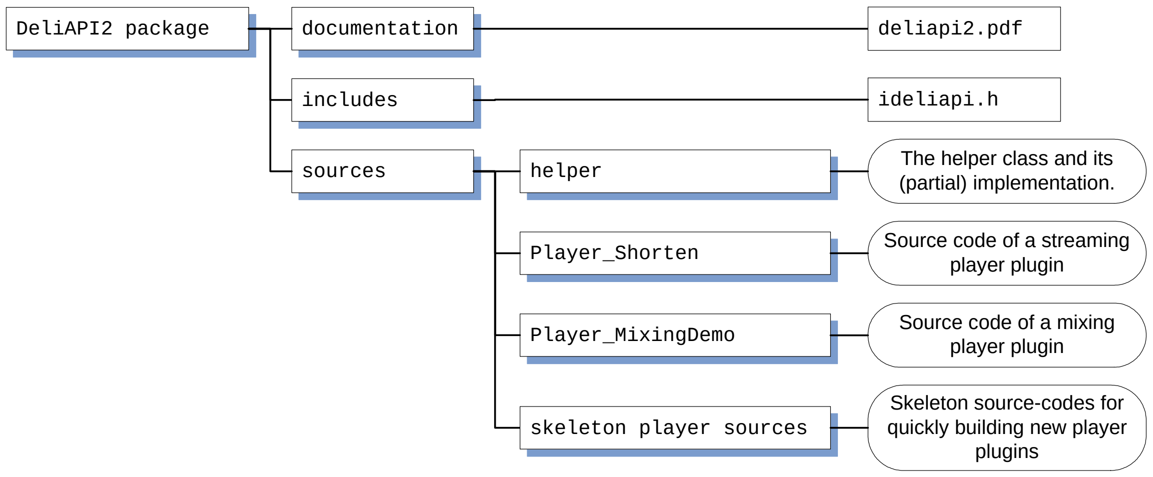
Creating software plugins can be easy and rewarding if the API of the application actually helps the plugin developer to build the necessary bridging code.
We think the DeliPlayer API achieves this through its straight forward design, the included helper classes, the plugin skeleton source, the included source code of two players and this documentation.
If, after reading the documentation, you have questions on how to use specific features of the API, if you are not sure on how to support a specific feature of the player you are adapting or have suggestions for future versions of the API, then please join the development forum on the DeliPlayer web site:

http://www.deliplayer.com/forum2/phpBB2/viewforum.php?f=9

You can also inquire there if someone else is already working on a replay that you plan to adapt.

Contents of the DeliAPI 2 package

The DeliAPI 2 package is a ZIP archive with the following directory structure:


DeliAPI2 package
documentation
deliapi2.pdf
includes
ideliapi.h
sources
helper
The helper class and its (partial) implementation.
Player_Shorten
Source code of a streaming player plugin
Player_MixingDemo
Source code of a mixing player plugin
skeleton player sources
Skeleton source-codes for quickly building new player plugins

The DeliAPI 2 does not contain or require a static link library (.lib). The API is described completely in the `ideliapi.h` header file. The helper class is included as source code.

Basics

The DeliAPI basically consist of three interfaces, `IDeliPluginSocketPlayer`, `IDeliPluginPlayer` and `IDeliPluginContainer`. Player plugins implement the `IDeliPluginPlayer` and `IDeliPluginContainer` interfaces. DeliPlayer provides an object that implements the `IDeliPluginSocketPlayer` interface to the each plugin to expose its API functions.
Player plugin developers using the provided helper classes only need to implement a subset of the `IDeliPluginPlayer` interface.

Architecture

- Plugins are standard Windows DLLs (Dynamic Link Libraries)
- One plugin DLL can contain one or more plugins.
- Uses an object-oriented design
- All API objects are COM objects. (see below why the plugin developer is not affected by this 'issue')

Special features of the DeliAPI 2

- Supports three methods for seeking in songs.
- Player plugins can use DeliPlayer's mixer to mix channels, they do not need to contain own mixing routines.

DeliPlayer plugins directly expose only one function through the DLL. This function (`GetDeliPluginContainer`) is called by DeliPlayer to retrieve a pointer to the container object of the plugin. The container object (Singleton) contains information about the plugin DLL and creates the plugin instances (Abstract Factory).
DeliPlayer then retrieves the types of the plugins this container supports. A container can contain one to several players. However, is it strongly recommended to create one plugin DLL per player.
DeliPlayer then creates several (at least two) instances per player class. Plugin developers must create thread-safe (reentrant) code because DeliPlayer uses the instances simultaneously.
Should it be impossible to write a thread-safe player because the code you use is not thread-safe and too complex to make thread-safe, then you can let DeliPlayer know that the plugin DLL contains non thread-safe code. In this case DeliPlayer will still create several instances of the player but will not use them simultaneously. As a consequence, DeliPlayer will not be able to cross-fade two tunes of this format, nor is it possible to pre-load a song in the format while another song is already playing.
Obviously it is highly recommended to write only thread-safe players, primarily because it is confusing for users why some formats behave differently than others.
Code that uses only one single global/static variable not purely const (read only) is not thread-safe. There is no such thing like 'mostly' thread-safe. The results of not declaring your thread-unsafe code as such range from erratic replay to being completely unstable.

The container object as well as the player objects are COM (Component Object Model) objects. This means they derive from the `IUnknown` interface and all COM programming rules apply to them. The container and player interfaces are defined in the `ideliapi.h` header file.
Included in the DeliAPI is a helper class that completely hides the COM nature of DeliPlayer's API so you can focus on writing the player code itself.
After a player object is instantiated, it receives a pointer to an object that implements the `IDeliSocketPlayer` interface which contains all functions that DeliPlayer exposes to its plugins. The socket contains functions to

- Extract data from DeliPlayer's file handles (`DELIFile`)
- Load secondary files
- Replay functions, including functions to use DeliPlayer's mixer

How to develop a player plugin

First of all, you don’t have to deal with the ‘gory’ details of the DeliPlayer API. Included in the DeliPlayer API is the C++ source code of a helper class that takes care of all COM related tasks. The API also contains two example players that demonstrate how mixing and streaming players work.
We highly recommend to always use the helper class from the DeliAPI package, because then your plugin is much more likely to remain source level compatible with later versions of the DeliAPI.

Using the supplied skeleton source as starting point for your player has the following advantages:

1. the player will compile, link and be loadable in DeliPlayer from the beginning on (provided the code you write is ok)
2. the skeleton imposes no overhead to your plugin (neither in size nor CPU time)
3. the developer does not have to deal with or even be aware of COM programming rules
4. the skeleton contains comments that help you understand the purpose of each method and its arguments
5. by following the skeleton’s TODO comments the developer is guided through the implementation process

Basic steps for implementing a player plugin

This section describes the basic steps needed to create a player plugin for DeliPlayer.

Microsoft Visual C++ 6.0 is used as development environment because it is widely spread. If you want to use another compiler/development environment you can of course do that, but you will have to figure out yourself if your compiler/IDE contains a project wizard that creates MFC application DLL projects and how to use it. If you use Microsoft Visual C++ 7.x (.NET 2003) the steps are very similar.
Because we want the player to have a configuration dialog (window) we use the MFC (Microsoft Foundation Classes) framework.

a. Start by creating the project for the player plugin and adding the skeleton plugin files to the project:
    - Launch the development environment (e.g. Visual C++ 6.0).
    - Open the ‘File’ menu and click ‘New’, click on the ‘Projects’ tab in the opened window and select *MFC AppWizard (DLL)*.
    - Enter the project name (something like ‘DP_Player_XYZ’), modify the position of the project as you see fit and press ‘OK’. The default settings in the next dialog are OK, so continue with ‘Finish’ and acknowledge with ‘OK’.
    - Open a file manager (e.g. Windows Explorer, Total Commander) and copy the skeleton player files from the DeliAPI package to the directory of the project you just created.
    - In Visual C++ open the ‘Project’ menu, click ‘Add to Project / Files’ and select all skeleton source files (*.cpp and *.h) that you have copied into the project directory in the previous step.

The project should now compile, link and already be loadable by DeliPlayer.

b. Once the project is set up and contains the skeleton player sources you have to fill in the actual player code. The skeleton and the helper class already implement all necessary interfaces for the player plugin. You now have to insert your code into the correct placeholders.
Insert/write the check code that recognizes music files in the particular format the player supports. (`Check()`)
    - Insert/write the initialization code that prepares the tune/plugin for playback. (`InitModule()` and `InitSong()` as well as `EndModule()` and `EndSong()`)
    - Insert/write the actual replay code. (`RenderFrame()`)

   At this point the player plugin should already load and play files in the supported format(s).

c. Adjust the `PluginInformation()`, `FormatInformation()` and `FileExtensionInformation()` implementations to provide DeliPlayer (and the user) with information about the player plugin itself (e.g. name, author, version and copyright).
Adjust the `SongInformation()`, `InstrumentInformation()`, `SampleInformation()` and `MessageInformation()` implementations to expose as much information as possible about the currently playing tune (e.g. song name, instrument names, author)

d. Decide which settings your player needs to expose to the user. You should expose as few (none) settings as possible because it is often hard to understand the effects of player settings unless you have actually written the player.
Implement the `ConfigLoad()`, `ConfigSave()`, `ConfigApply()`, `ConfigDefaults()` and `ConfigChanged()` functions.

e. Design the configuration dialog and write the code to initialize the dialog controls and retrieve their content.
    - Add a dialog resource to the project via the menu *Insert/Resource*.
    - The first thing you should do is right click into the dialog and open its Properties window from the context menu. Go to the *Styles* tab, set the *Style* to ‘Child’, un-check the *Title bar* checkbox and select ‘None’ in the *Border* combobox.
    - Add all required controls to the dialog (do not forget to verify/adjust the tab-order (see menu *Layout/Tab Order*)).
    - Open the *MFC ClassWizard*, which should immediately ask if you want to create a new class for the dialog, which you should do. Give the class an explanative name like `CConfigDlg` an exit with OK.
    - In the *MFC ClassWizard* create *Message Maps* and *Member Variables* (variable bindings) as you see fit, and add the necessary code to initialize the controls and retrieve their content.
    - Implement the `ConfigDialogOpen()`, `ConfigDialogClose()`, `ConfigDialogApply()` and `ConfigDialogRestore()` functions.

f. Debugging and testing. You should be especially careful about the check routine and sanity checks of the files you are loading.
    - Ten fold check your check routine. Don’t laugh, do it. DeliPlayer already supports several hundred formats amongst which are some formats that have proven to possess qualities of a chimera. Many users add directories with lots of potential music files and rely on DeliPlayer to pick out those files that are actually supported. Your player will very likely come across some files that look very much the files in your format but are actually something completely different. Very small and very big files are also potential problems. To avoid unpleasant surprises you should make your check routine as strong as possible and test it thoroughly.
      Do not even think about relying on file extensions (e.g. .mod) to identify a certain format. Your player is guaranteed to crash or behave very weird each time a file is loaded that accidentally has the extension you check for but is not in the format you expect.
    - The more popular the format your player supports is, the more likely it is that you will have to deal with severely damages files, incomplete files and files saved by slightly modified versions of the editor software. If in doubt, better reject a file first instead of crashing later.

If your player does not need a configuration dialog it is still a good idea to create a project that supports MFC so if you later decide to implement a configuration dialog you can do this without having to create a new project. Also, the MFC framework provides several useful types like `CString`, `CList` and `CArray`.

Implementing the replay code

Allocating and initializing channels

Before you can play sound using the `ReplaySetSample()`/`ReplaySetRepeat()` functions, you have to call the `InitReplay()` function first. Besides allocating a given number of (virtual) channels, this function does also some other important things:

- The number of channels depends on the format (and files) you want to play. If a format supports stereo samples, it is recommended to allocate only one channel and to play stereo samples on it instead of splitting a stereo sample into two mono samples.
- The `maxvolume` and `maxpanning` parameters specify the maximum volume and panning values the player uses. Volume values higher than the `maxvolume` parameter are clipped to `maxvolume`. Setting `maxpanning` to zero disables variable panning for this player. Both arguments help DeliPlayer to compute the overall sound output level of all channels properly.
- Players can operate in either period or frequency mode. If you specify a `periodbase` of zero, the arguments of a ReplaySetFrequency() call are treated as frequencies. Otherwise, the sample playback rate (in Hz) can be calculated as follows:
  ```
  playback_rate := period_base / setfrequency_argument
  ```
- DeliPlayer distinguishes between mixing and streaming players. Mixing players are players where the instruments (which usually play at different frequencies) have to be mixed to single output stream playing at e.g. 44100Hz. All tracker players fall into this category. The speed at which the `RenderFrame()` routine of a player is called can be specified in two ways for mixing players:
  1. if `timerbase` is not zero, the `timerspeed` parameter acts as a divider for the `timerbase`. The following formula can be used to compute the number of calls which occur per second:
     ```
     renderframe_rate := timer_base / timer_speed
     ```
  2. if `timerbase` is set to zero, the `timerspeed` value specifies the ProTracker BPM tempo at which the player is called

  In either case, the `ReplaySetSpeed()` function can be used during playback to modify the current playback speed.

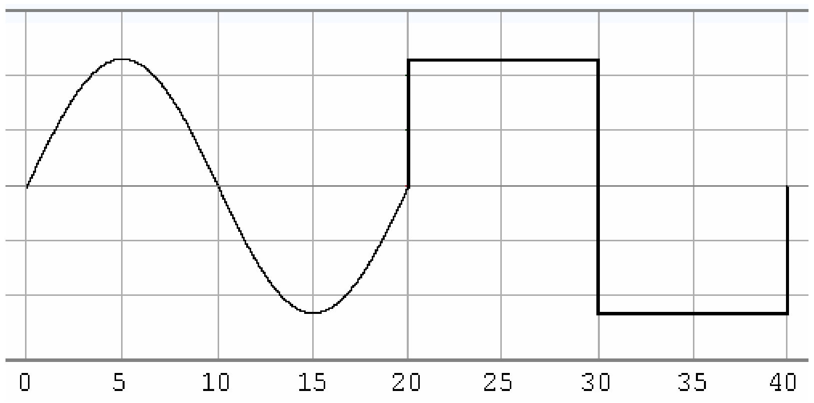
  If both `timerbase` and `timerspeed` are set to zero, a player operates in streaming mode. This mode is used for formats where the player itself generates the output samples in real-time or when the samples are loaded from disk. The rate at which the player is called is then determined by the length and the frequency of the playing samples. As a consequence, all playing samples must have the same length and frequency if several samples are played simultaneously in streaming mode. Furthermore, it is recommended to split the samples into chunks which have a length of approximately 1/100 to 1/10 second.

Once you have allocated some channels, it is recommended to initialize each channel to valid initial values by using the `InitChannel()` function. You can set the initial channel number (Mono/Stereo), resolution, frequency, volume and panning for an instrument playing on a certain channel. The initial vales are overwritten by a subsequent `ReplaySet*()` function calls.

Although it is recommended to allocate only the minimum number of channels needed for playing a certain song or sample, it is possible to allocate more channels and to disable the inactive channels. This can be done by setting the `numchannels` argument of `InitChannel()` to zero. If a channel is disabled, DeliPlayer ignores all `ReplaySetSample()` and `ReplaySetRepeat()` calls for this channel.

Playing samples

The DeliPlayer API splits samples into a one-shot part and into a repeat part (often also called loop part of a sample). See for example the following sample which consists of a leading sine part and a trailing rectangle part.



Let's say we want to play this sample with DeliPlayer. In the following, we use the sine part (time index 0 - 19) as one-shot, and the square part (time index 20 - 39) as repeat part. Let us further assume that the sample is located at offset `0x123456` in memory, is signed, mono and has a resolution of 8 bits. Then the function call plays the sample on the output channel 3:

```
ReplaySetSample( 3, 1, 8, 0x123456, 20 );
```

Note that we have not set the loop part of the sample yet. If we omit the following ReplaySetRepeat() call, DeliPlayer plays only the one-shot part specified above, then the sample stops. To play the square part in a looped way, we make the following call:

```
ReplaySetRepeat( 3, 1, 8, 0x123456+20, 20 );
```

Because the loop part starts 20 bytes after the one-shot part, we add 20 to the base address of the sample. The size of the loop is again 20, so we use 20 as repeat length (all other sample attributes like the channel number and the resolution are assumed to be identical).

Some remarks about the `ReplaySetSample`/`ReplaySetRepeat` functions:

- Mono and stereo samples are supported. Set the `numChannels` argument to `1` for mono samples, for stereo samples set it to `2`. In case of stereo samples, the sample data must be interleaved, starting with the left channel (i.e. left0, right0, left1, right1, and so on). Samples which have a `numChannels` argument of zero are not mixed.
- The sample data must be signed (if the samples of the module/sound format you're planning play are unsigned, you have to convert them to signed format first).

- The sample resolution (i.e. the bits per sample) can be either 1 - 16 bit for integer samples or 32 bit for float samples. 1 - 8 bit samples have to be in signed char format (i.e. one byte per sample), 9 - 16 bit samples have to be in signed short format (i.e. two bytes per sample, Intel byte ordering). `float` samples have to be normalized to the range [-1.0..+1.0]. For integer samples, simply supply the bits per sample value as resolution. For float samples, supply zero as resolution.
- Both the length of the one-shot part and the length of the repeat part of a sample have to be specified in samples, not bytes. In case of stereo samples, the length has to be specified as number of stereo sample pairs.
- While a sample is playing, the repeat part of the sample can be modified by subsequent `ReplaySetRepeat()` function calls. Changing the loop part of a sample usually doesn't have an effect immediately. Instead, DeliPlayer completely plays the current part (either one-short or repeat). Once this part as reached the end, the playback of the newly set repeat part starts. If DeliPlayer should immediately play the specified looped sample you have to additionally call `ReplaySetSample()` with the same sample pointer and length.

To actually play a song, calling `ReplaySetSample` or `ReplaySetRepeat` is usually not enough. You also have to set the frequency (i.e. playback rate) and the volume of the sample as well. Often, a `ReplaySetFrequency`/`ReplaySetVolume` call occurs along with `ReplaySetSample`. Note that a frequency of zero stops a playing sample when in frequency mode and setting the frequency to a value greater than zero resumes the sample from the stopped point. The volume of the sample can be set with the `ReplaySetVolume`() function. The panning function calls are optional. When not used, a sample is played at the center position.

Loading secondary files

Players sometimes need to load secondary files that belong to the module that the user wants to play. See `FileLoad()` and `FileGetData()`. Loading of additional files should be done in `InitModule()` because this function is called exactly once for each module. Player should always use the file operation functions from the DeliAPI because DeliPlayer handles archive extraction and decompression on its own. If for example the user plays a tune from a ZIP archive and the player needs to load a secondary file (also contained in the ZIP archive) then using the DeliAPI file functions automatically extracts the file from the archive, whereas the player would have a quite difficult time doing the same on its own.
If a player needs to load additional files that do not belong to a module, e.g. static tables and configuration files, then `FileLoad()` can not be used because `FileLoad()` can only be called in or after `InitModule()` and the loaded files are automatically unloaded after `EndModule()`. Use `fopen()`, `CreateFile()`, `ifstream` or similar instead.

Seeking

Seeking means randomly changing the playback position.
DeliPlayer supports three different techniques for players to implement seeking functionality.
The seeking techniques should be used mutually exclusive (meaning: implement none or one of these techniques):

- Implement the `SetPosition()` function. Usually used by streaming players.
- Implement the `StartSkip()` and `StopSkip()` functions. Usually used by players that mix themselves.
- Implement the `replay state` block. Usually used by mixing players.

SetPosition

The normal (and easiest) way to support seeking in streaming players is to implement the `SetPosition()` function. In your implementation you call the `seek` function of the player's decoder with the position argument of `SetPosition()`.
Implementing `SetPosition()` in mixing players or streaming players that do not have a `seek` function can be a lot of hard work or be impossible.

StartSkip and StopSkip

DeliPlayer calls the players `StartSkip()` function to tell it that `RenderFrame()` should do as little as possible and just increment the playing position until `StopSkip()` is called. Frames rendered in between `StartSkip()` and `StopSkip()` calls are not used for playback.
DeliPlayer 'skips' through the song until the desired position is reached and then turns off skip mode.

Replay State Block

The replay state block technique usually can be used only with thread-safe players whose replay engine is well understood. It is the most dangerous of the three techniques because the player developer must be very careful when implementing it. However, it is also the most elegant solution, at least technically.
This technique requires the player to group together all variables that are modified during the replay of a song in one structure, which is called the `replay state` block. The player then exposes the size and address of its replay state block. During playback DeliPlayer creates backup copies of the replay state block in certain intervals (about 10 seconds). When the playing position is changed, DeliPlayer checks which backup copy is closest before the desired position, copies the backup copy onto the players replay state block and calls the players `RenderFrame()` function until the desired position is actually reached. While skipping to the desired position DeliPlayer does not mix the frames.
The replay state block should not be larger than 30 kBytes, otherwise the player should use the `StartSkip()/StopSkip()` technique or not implement seeking.

Data types used by the DeliAPI 2

The DeliAPI 2 uses only types used in the C calling convention. A few typedefs and structures are defined to improve readability, type safety and later source level compatibility.

DeliTag

A DeliTag represents one entry in a tag array. Tag arrays are arrays of {id, value} pairs, where the id describes the meaning of the next value. Tag arrays are terminated by a {0,0} entry.

DELITagId

The id in a DeliTag.

DELITagValue

The value in a DeliTag.

DELIBool

The bool data type. Allowed values are `DELI_TRUE` and `DELI_FALSE`.

DELIFile

A handle to a DeliAPI file.

DELIResult

The data type returned by DeliAPI 2 functions and plugin functions.

DELIVersion

The data type used for storing version information.

DeliNamedVariable

A DeliNamedVariable represents one entry in the array that describes the variables a plugin exposed to DeliPlayer. (currently unused)

DeliNamedVariableValue

DeliNamedVariableValue is a union used to access the different data types a named variable exposed by DeliAPI plugin can have (currently unused).

Player plugin interfaces

DeliPlayer player plugins contain implementations of the `IDeliPluginContainer` and `IDeliPluginPlayer` interfaces.

Overview:

`GetDeliPlayerPluginContainer`	Returns the container object. (Singleton)
IDeliSocketPlayer	
`Initialize`	Perform internal initialization
`Information`	Return information about the plugin container
`GetIID`	Return the IID of the plugin at the specified index position
`CreateInstance`	Instantiates a plugin (Abstract Factory)
`Cleanup`	Deinitialize the plugin container object.
IDeliPluginPlayer:	
`InitPlugin`	Initializes the player instance
`Check`	Check if a file is in a supported format
`InitModule`	Initializes the player to start playing a file
`InitSong`	Set the sub-song index to play
`StartPlay`	Start the playback
`StartSkip`	Activate skip mode
`RenderFrame`	Render one frame of audio/music data
`SetPosition`	Jump to a different playback position
`EndSkip`	Deactivate skip mode
`EndPlay`	Stop playback
`EndSong`	Deinitialize the song
`EndModule`	Deinitialize from playback
`EndPlugin`	Deinitialize the plugin
`PluginInformation`	Provides information about the player, e.g. Name, Copyright
`SetVariable`	Modify the specified plugin variable with the provided value
`GetVariable`	Return the current value of the specified plugin variable
`SetVariablesApplyMode`	<currently unused>
`Notify`	<currently unused>
`ConfigDialogOpen`	Open the configuration dialog for the player
`ConfigDialogClose`	Close the configuration window
`ConfigDialogApply`	Apply the settings from the config. dialog to the player
`ConfigDialogRestore`	Reset config. dialog to the settings of the player
`ConfigDefault`	Reset the settings of the player to their default values
`ConfigApply`	Apply the settings of another instance
`ConfigSave`	Save the configuration file
`ConfigLoad`	Load the configuration file
`ConfigChanged`	Return true if the configuration was changed
`SongInformation`	Return information about a song
`SampleInformation`	Returns information about a sample
`InstrumentInformation`	Return information about an instrument
`MessageInformation`	Return a song-message

GetDeliPlayerPluginContainer

GetDeliPlayerPluginContainer is the only function that is directly exposed through the DLL. DeliPlayer calls it to retrieve a pointer to the container object of the plugin DLL (Singleton).

IDeliPluginContainer

Initialize

Performs internal initialization that is needed only once for all plugin instances like loading additional libraries and pre-calculating static tables.
Returns:

`DELIResult`	Success or failure

Parameters: none

Cleanup

Called only if the previous call to `Initialize()` succeeded.
Free all resources allocated in the last call to `Initialize()`
Returns: `void`
Parameters: none

Information

Return information about the plugin container. Currently this tag-array uses one single entry. `DATA_Container_NumPlugins` providing the number of different plugin (player) classes the container supports.
It is, however, recommended to implement only one plugin (player) class per container.
Returns:

`DeliTag*`	Pointer to the tag-array containing information about the container

Parameters: none

GetIID

Called only if the previous call to `Initialize()` succeeded.
Returns the IID (Interface IDentifer) of the interface that the plugin (player) object at index `pluginindex` implements.
It is recommended to implement only one (plugin) player class per DLL, so your implementation of `GetIID()` would return `IID_IDeliPluginPlayer` for `pluginindex==0`.
Returns:

`REFIID`	The IID

Parameters:

`int pluginindex`	

CreateInstance

Create an instance of the player class at index `pluginindex`. The REFIID parameter explicitly specifies the interface the plugin (player) object must implement. If the class at index `pluginindex` does not implement the requested interface, `CreateInstance()` should return `DELI_RESULT_ERROR_FAILED`.

Returns:
`DELIResult` Success or failure
Parameters:

`int pluginindex`	The index of the requested plugin class
`REFIID`	The IID of the requested object
`PPVOID`	Pointer to a void pointer, into which your implementation copies the pointer to the created instance.

IDeliPluginPlayer

InitPlugin

Called once per player instance.
Initialize the player instance. The `pSocket` parameter points to an object implementing the `IDeliPlayerSocker` interface through which the player calls functions in DeliPlayer. It should be stored for later use. The object remains valid until after `EndPlugin()` is called.
Returns:
`DELIResult` Success or failure
Parameters:

`PVOID pSocket`	Pointer to the socket object.

EndPlugin

Called only if the previous call to `InitPlugin()` succeeded.
Deinitialize the player instance and free all resources allocated in and since the previous call to `InitPlugin()`.
Returns: `void`
Parameters: none

PluginInformation

Called only if the previous call to `InitPlugin()` succeeded.
Called to retrieve information about the player.
Returns:
`DeliTag*` The tag-array containing information about the plugin.
Parameters: none

SetVariable

Currently unused
Returns:
`DELIResult`
Parameters:

`int varid`	
`DeliNamedVariableValue Value`	

GetVariable

Currently unused
Returns:
`DELIResult`

Parameters:

`int varid`	
`DeliNamedVariableValue* pValue`	

SetVariablesApplyMode

Currently unused
Returns: void
Parameters:

`int`	

Notify

Currently unused
Returns: `void`
Parameters:

`unsigned int event`	
`unsigned int data1`	
`unsigned int data2`	

ConfigDialogOpen

Called only if the previous call to `InitPlugin()` succeeded.
Open the player's configuration dialog. The window should not have a border and no drag-bar because DeliPlayer will display the window inside its own configuration window.
Returns:
`HWND` The handle of the configuration window, or NULL in case of an error.
Parameters: none

ConfigDialogClose

Called only if the previous call to `ConfigDialogOpen()` succeeded.
Close the configuration dialog and free all resources allocated in `ConfigDialogOpen()`
Returns: `void`
Parameters: none

ConfigDialogApply

Called only if the previous call to `InitPlugin()` succeeded.
Apply the settings from the configuration dialog to the player.
Returns: `void`
Parameters: none

ConfigDialogRestore

Called only if the previous call to `InitPlugin()` succeeded.
Set the configuration dialog to the player's current settings.
Returns: `void`
Parameters: none

ConfigDefault

Called only if the previous call to `InitPlugin()` succeeded.
Reset the player's settings to the default values.
Returns: `void`
Parameters: none

ConfigApply

Called only if the previous call to `InitPlugin()` succeeded.
Apply the settings of another instance of the same player to this instance of the player. If your player has a configuration dialog you also should implement the `ConfigApply()` function otherwise only one instance of your player can be re-configured.
Returns: `void`
Parameters:

`void* newcfg`	A pointer to the settings data of another instance of the same player

ConfigSave

Called only if the previous call to `InitPlugin()` succeeded.
Save the settings of the player. Players usually store their settings in a file in the current directory (which DeliPlayer sets to the appropriate location). There are currently no rules on how players should store their settings.
Returns:

`DELIResult`	Success or failure

Parameters: none

ConfigLoad

Called only if the previous call to `InitPlugin()` succeeded.
Load the settings of the player.
Returns:

`DELIResult`	Success or failure

Parameters: none

ConfigChanged

Called only if the previous call to `InitPlugin()` succeeded.
Called by DeliPlayer to check if the settings of the player have changed since the last call to `ConfigLoad()` or `ConfigSave()`.
Returns:

`DELIBool`	`DELI_TRUE` if the settings have changed, `DELI_FALSE` otherwise.

Parameters: none

Check

Called by DeliPlayer for every file that is loaded. If your check code identifies the file and the plugin can play it, return `DELI_RESULT_OK`. If the file was identified but is corrupt (e.g. too short), return `DELI_RESULT_ERROR_FORMATCORRUPT.` If the file is an unsupported sub-format, return `DELI_RESULT_ERROR_FORMATNOTSTANDARD`, otherwise return `DELI_RESULT_ERROR_FAILED`.
Never store anything retrieved via the `DELIFile` argument (i.e. data pointers, size)

Returns:

`DELIResult`	Valid Values:	`DELI_RESULT_OK`
		`DELI_RESULT_ERROR_FAILED`
		`DELI_RESULT_ERROR_FORMATNOTSTANDARD`
		`DELI_RESULT_ERROR_FORMATCORRUPT`

Parameters:

`DELIFile`	The file to check

InitModule

Called only if the previous call to `Check()`succeeded.
Initializes the plugin/provided data as needed to start playback. The `primaryfile` remains valid until after `EndModule()`or until it is explicitly unloaded by the player. The content of the `primaryfile` (retrieved with `FileGetData()`) can be modified by the player.
Returns:

`DELIResult`	Valid Values:	`DELI_RESULT_OK`
		`DELI_RESULT_ERROR_FAILED`
		`DELI_RESULT_ERROR_FORMATNOTSTANDARD`
		`DELI_RESULT_ERROR_FORMATCORRUPT`

Parameters:

`DELIFile primaryfile`	The file to play

EndModule

Called only if the previous call to `InitModule()` succeeded.
Eject the file and reset the replay engine. The player should free all resources allocated in and since the previous `InitModule()` and `InitSong()` calls.
Returns: `void`
Parameters: none

InitSong

Called only if the previous call to `InitModule()` succeeded.
Initializes the replay engine to play the provided song index.
Returns:

`DELIResult`	Success or failure

Parameters:

`int songindex`	The index of the sub-song to play

EndSong

Called only if the previous call to `InitSong()` succeeded.
Deinitialize the engine from playing a sub-song. The player should free all resources allocated in and since the previous `InitSong()` call.
Returns: `void`
Parameters: none

StartPlay

Called only if the previous call to `InitSong()` succeeded .

Start playback. This method is usually unused, it however is is required to implement players that require their own thread or access/require (asynchronous) devices for playback (ie.: MIDI,CDDA).
Returns:
`DELIResult` Success or failure
Parameters: none

StopPlay

Called only if the previous call to `StartPlay()` succeeded.
The player should free all resources allocated in and since the previous `StartPlay()` call.
Returns: `void`
Parameters: none

StartSkip

Called only if the previous call to `StartPlay()` succeeded.
Switch the replay engine to skip mode. The functions should be implemented in players that do all mixing themselves. When in skip mode, the player's `RenderFrame()` function should do as little as necessary to proceed from one frame to the next frame and should not generate any output data.
Returns:
`DELIResult` Success or failure
Parameters: none

StopSkip

Called only if the previous call to `StartSkip()` succeeded.
Switch off skip mode.
Returns: `void`
Parameters: none

RenderFrame

Called only if all previous calls to the player's Init functions succeeded.
Render one frame of audio/music data. A 'frame' usually covers between 1/100 to 1/10 second. It should never exceed 2 seconds.
Returns:
`DELIResult` Success or failure
Parameters: none

SetPosition

Called only if all previous calls to the player's Init functions succeeded.
Set the playback position `dPosition` is a floating point value containing the desired playback position in seconds, with at least millisecond precision.
Returns:
`DELIResult` Success or failure
Parameters:

`double dPosition`	The time position (in seconds) to jump to

SongInformation

Called only if the previous call to `InitModule()` succeeded.
Return a tag-array containing information about a specific sub-song of the currently loaded module. The `songindex` argument specifies the sub-song for which information is requested. An index of -1 is used to get module global (sub-song unspecific) information. Players/formats that do not support sub-songs (like most streaming formats) should return a tag-array only for `songindex` arguments -1 and/or 0.
Players should return as much information as possible, so don’t be lazy in your implementation.
Returns:

`DeliTag*`	Pointer to a tag-array, or NULL in case of an error.

Parameters:

`int songindex`	The index of the song for which information is requested

InstrumentInformation

Called only if the previous call to `InitModule()` succeeded.
Return a tag-array containing information about am instrument of a specific sub-song of the currently loaded module. See also SongInformation() for an explanation of the `songindex` argument. DeliPlayer will call this function with incrementing `instrumentindex` argument until it returns NULL to indicate that the requested instrument does not exist.
Returns:

`DeliTag*`	Pointer to a tag-array, or NULL in case of an error.

Parameters:

`int songindex`	The index of the song for which information is requested
`int instrumentindex`	The index of the instrument for which information is requested

SampleInformation

Called only if the previous call to `InitModule()` succeeded.
Return a tag-array containing information about a sample of a specific sub-song of the currently loaded module. See also SongInformation() for an explanation of the `songindex` argument. DeliPlayer will call this function with incrementing `sampleindex` argument until it returns NULL to indicate that the requested sample does not exist.
Returns:

`DeliTag*`	Pointer to a tag-array, or NULL in case of an error.

Parameters:

`int songindex`	The index of the song for which information is requested
`int sampleindex`	The index of the sample for which information is requested

MessageInformation

Called only if the previous call to `InitModule()` succeeded.
Return a tag-array containing the message. See also SongInformation() for an explanation of the `songindex` argument. DeliPlayer will call this function with incrementing `messageindex` argument until it returns NULL to indicate that the requested message does not exist.
Returns:

`DeliTag*`	Pointer to a tag-array, or NULL in case of an error.

Parameters:

`int songindex`	The index of the song for which information is requested

`int messageindex`	The index of the message for which information is requested

FormatInformation

Called by DeliPlayer to retrieve information about the formats supported by this player. DeliPlayer will call this function with incrementing `formatindex` argument until it returns NULL to indicate that the requested format does not exist. If a player supports more than one format (with the same replay engine) then it should return one format tag array for each (sub) format supported.

Returns:

`DeliTag*`	Pointer to a tag-array, or NULL in case of an error.

Parameters:

`int formatindex`	The index of the song for which information is requested

FileExtensionInformation

Called by DeliPlayer to retrieve information about the file extensions supported by this player. DeliPlayer will call this function with incrementing `extindex` argument until it returns NULL to indicate that the requested file extension does not exist.
DeliPlayer will offer the user to register the file extensions the player exposes. You should be careful not to expose file extensions (although belonging to the format your player supports) that are known to be used by widely spread applications so users can not accidentally disable the *click to launch* behavior they are used to/expect.

Returns:

`DeliTag*`	Pointer to a tag-array, or NULL in case of an error.

Parameters:

`int extindex`	The index of the song for which information is requested

IDeliSocketPlayer interface

The `IDeliSocketPlayer` interface describes the functions DeliPlayer exposes to its player plugins. Players access them through the socket object provided by DeliPlayer in the call to the player’s `InitPlugin()` function.

Overview:

`InitReplay`	Initialize the replay engine
`InitChannel`	Initialize a specific channel
`SetStateData`	Set the pointer to the players replay state block
`GetStateData`	Retrieve the replay state block set earlier
`SetConfigData`	Set the pointer to the player configuration data
`GetConfigData`	Retrieve the player configuration data pointer set earlier
`GetOutputFrequency`	Retrieve the current output frequency
`ReplaySetSpeed`	Change the replay speed
`ReplayNotifySongEnd`	Notify the core that the end of the song was reached
`ReplaySetSample`	Start playing a sample on a channel
`ReplayStopSample`	Stop the currently playing sample on the channel
`ReplaySetRepeat`	Initializes a looped sample on the channel.
`ReplayFinishRepeat`	Stops playing the loop part
`ReplaySetVolume`	Set the volume on a channel
`ReplaySetPan`	Change the panning on the channel
`ReplaySetFrequency`	Set/change the sample frequency on the channel
`ReplayInvertPlayingDirection`	Invert the playing direction of the loop part
`FileLoad`	Completely load a file into memory
`FileSeek`	Seek to a new position in a file
`FileRead`	Partially read the content of a file
`FileTell`	Retrieve the current absolute position in a file
`FileGet`	Retrieve a `DELIHandle`
`FileAlloc`	Create an in-memory ‘file’
`FileUnload`	Close a file and free its memory
`FileGetData`	Retrieve the files content
`FileGetName`	Retrieve the name and path of the file

InitReplay

Call InitReplay in your implementation of InitModule() or InitSong(). It initializes the replay and mixer engines in DeliPlayer.

Returns:

`DELIResult,` Indicates success or failure

Parameters:

`unsigned int nChannels`	The number of channels used by the loaded song
`int maxFrequency`	The maximum frequency
`int maxVolume`	The maximum volume
`int maxPanning`	The maximum panning
`int periodBase`	See here
`int timerBase`	The initial speed, see here
`int timerSpeed`	The initial speed, see here
`unsigned int flags`	

InitChannel

Call `InitChannel()` to set a channel to a default state. This function should be called in `InitModule()` or `InitSong()`.

Returns: `void`

Parameters:

`unsigned int iChannel`	The index of the channel that should be initialized
`int numchannels`	The number of channels (mono/stereo) Valid values: 1, 2
`int resolution`	The sample resolution. Valid values: 0 (float sample), 1 – 16 (integer sample)
`int frequency`	The initial replay sample rate
`int volume`	The initial volume
`int horizpan`	The initial horizontal (left/right) panning
`int depthpan`	The initial depth (front/back) panning

SetStateData

Call SetStateData() to set the replay state block pointer. If your player supports the replay state block feature, you should call SetStateData() in your implementation of the InitPlayer() function. Usually used only by mixing players.

Returns: `void`

Parameters:

`void* statedata`	The pointer to the replay state block of the player plugin

GetStateData

Retrieve the pointer to the replay state block your player set earlier.

Returns:

`void*` The pointer to your player's replay state block.

Parameters: none

SetConfigData

Call SetConfigData() to set the pointer to your players internal configuration structure. DeliPlayer handles this pointer without knowing anything about the structure/implementation of the player's configuration.
If your player does not have a configuration, SetConfigData() should not be called. If your player has a configuration window/structure, SetConfigData() should be called otherwise only one instance of your player can be re-configured with its config dialog. This is necessary because DeliPlayer works with at least two instances of every player plugin. To avoid forcing the user to re-configure each instance of the plugin, DeliPlayer calls the ConfigApply() function of every instance with the pointer to the configuration structure of the one instance whose dialog was actually changed by the user, so these instances can also apply the modified configuration.

Returns: `void`
Parameters:

`void* configdata`	The pointer to the config data of the player plugin

GetConfigData

Retrieve the pointer to the configuration structure your player set earlier.
Returns:
`void*` The pointer to your player's replay state block.
Parameters: none

GetOutputFrequency

Retrieve the current output sample rate. If your player requires the output sample rate you should call GetOutputFrequency in your implementation of InitModule() or InitSong().
The output sample rate is guaranteed to be constant between the calls to the player's InitModule() and EndModule() functions. Usually used only by players that do all mixing themselves.
Returns:
`int` The output sample rate (frequency).
Parameters: none

SetTotalFileSize

Call SetTotalFileSize() after your player has loaded all its secondary files. The total file size is not used by DeliPlayer, it is an informational value that can be displayed in the user interface.
If your player uses DeliPlayer's File API to load secondary file, you do not need call SetTotalFileSize(), you can however call it to override the internally calculated value.
Returns: `void`
Parameters:

`double nBytes`	The total size (in bytes) of all files belonging to the currently loaded song. The `double` type is used to allow setting values greater than (2^32)-1

SetTotalRealSize

Call SetTotalRealSize() after your player has loaded all its secondary files. The total real size is not used by DeliPlayer, it is an informational value that can be displayed in the user interface. If your player uses DeliPlayer's File API to load secondary file, you do not need call SetTotalRealSize() unless your player further decompresses the files belonging to the currently playing song. You can call it to override the internally calculated value.
Returns: `void`
Parameters:

`double nBytes`	The total size (in bytes) of all decompressed files belonging to the currently loaded song. The `double` type is used to allow setting values greater than (2^32)-1

ReplayInformationChanged

Call ReplayInformationChanged() if a info value changes while playing the song. This function is usually used only by streaming players to broadcast a changed bitrate for variable bitrate streams.
Returns: `void`
Parameters:

`int eWhat`	Specifies which value has changed
`unsigned int value`	The new value

ReplaySetSpeed

Call ReplaySetSpeed to change the replay speed during replay for mixing players. Do not call this function in streaming players. `ReplaySetSpeed()` actually increases/decreases the size of a sample frame. Using an argument of zero is not allowed.
Returns: `void`
Parameters:

`int speed`	The new `timerspeed` value. Must not be 0. See also InitReplay.

ReplayNotifySongEnd

Call ReplayNotifySongEnd() from within your implementation of RenderFrame() when the player reaches the end of the song while processing the frame.
Returns: `void`
Parameters:

`int songendflags`	The location of the songend. Valid values: 0, `DELI_SONGEND_ATENDOFFRAME`

ReplaySetSample

Call ReplaySetSample to start playing the one-shot part of an instrument on a channel. Used only by mixing players. Streaming player only use ReplaySetRepeat().
Returns: `void`
Parameters:

`int iChannel`	The index of the channel on which to play the instrument
`int numChannels`	The number of channels used (stereo/mono). Valid values: 1,2
`int resolution`	The sample resolution. Valid values: 0,1-16

`void* pSamples`	The pointer to the start of the sample array
`unsigned int nSamples`	The number of samples in the sample array (one stereo sample counts as one sample)

ReplayStopSample

Call ReplayStopSample() to stop playing the sample (one-shot as well as repeat part). This function immediately silences the channel.
Returns: `void`
Parameters:

`int iChannel`	The index of the channel

ReplaySetRepeat

Returns: `void`
Parameters:

`int iChannel`	The index of the channel
`void* pSamples`	The pointer to the sample array.
`unsigned int nSamples`	The number of samples in the sample buffer (one stereo sample counts as one sample)
`int looptype`	The loop type. Valid values: DELI_LOOPTYPE_Forward, DELI_LOOPTYPE_Backward, DELI_LOOPTYPE_PingPong

ReplayFinishRepeat

Call ReplayFinishRepeat() to stop playing the loop-shot part the next time it would be repeated.
Returns: `void`
Parameters:

`int iChannel`	The index of the channel

ReplaySetVolume

Call ReplaySetVolume() to set the volume of a channel.
Returns: `void`
Parameters:

`int iChannel`	The index of the channel
`int volume`	The new volume

ReplaySetPan

Call ReplaySetPan() to set the horizontal (left/right) and depth (front/back) pan position of a channel. Currently, depth panning can only be used to play on the surround speaker by setting `horizpan` to 0 and `depthpan` to the `maxpanning` value specified in the InitReplay() call.
Returns: `void`
Parameters:

`int iChannel`	The index of the channel
`int horizpan`	The new horizontal (left/right) pan position
`int depthpan`	The new depth pan (front/back) position

ReplaySetFrequency

Call ReplaySetFrequency() to change the sample rate of a channel.

Returns: `void`
Parameters:

`int iChannel`	The index of the channel
`int frequency`	The new sample rate (i.e. period or frequency)

ReplayInvertPlayingDirection

Call ReplayInvertPlayingDirection() to change playing direction of the loop-part of a sample from forward to backward or vice versa.
Returns: `void`
Parameters:

`int iChannel`	The index of the channel

FileLoad

Call FileLoad() to load a file into memory. If the file does not contain an absolute path, DeliPlayer tries to load it from the same directory or archive in which the primary file is located. If the file is compressed with a known compressor, it will automatically be decompressed. To access the content of the file call FileGetData().
Returns:

`DELIFile`	A handle to the loaded file, or NULL in case of an error.

Parameters:

`const char* filename`	The name of the file to load

FileSeek

Call FileSeek() to seek to a new position in a file. Usually used only by streaming players.
Returns:

`double`	The new absolute position in the file.

Parameters:

`double newposition`	The position to seek to
`int disposition`	The seek disposition. Valid values: `DELI_SEEKORIGIN_BEGIN`, `DELI_SEEKORIGIN_CURRENT`, `DELI_SEEKORIGIN_END`

FileRead

Call FileRead() to partially read a file. Used only by streaming players.
Note: Due to the absence of a FileOpen() function, you currently can use FileRead() only with the primary file.
Returns:

`DELIResult`	Success or failure.

Parameters:

`DELIFile file`	The file to read from

FileTell

Call FileTell() to retrieve the current absolute position of a `DELIFile`. Usually used only by streaming players.
Returns:

`double`	The current absolute position in the file.

Parameters:

`DELIFile file`	The file of which to tell the position

FileGet

Call FileGet() to the `DELIFile` handle of the file at a specific index position. Index 0 is the primary file.
Returns:

`DELIFile`	The handle to the file, or NULL in case of an error.

Parameters:

`int index`	The index of the file

FileAlloc

Call FileAlloc() to create a new ‘file’ in memory. This mechanism is usually used by players to convert files in memory (decompress, re-structure). Call FileUnload() to free the file when the player no longer needs it. Note that DeliPlayer automatically frees all `DELIFiles` after the EndModule() function of the player has been called.
Returns:

`DELIFile`	The handle to the new in-memory file.

Parameters:

`int buffersize`	The index of the file
`const char* filename`	The name of the file. Not used by DeliPlayer.
`int wallsize`	The size of a ‘no mans land’ area (*wall*) behind the file. Using a wall can be useful e.g. for decompressing files when the decompressor temporarily requires additional space in the destination buffer, or for security when dealing with too short files.

FileUnload

Call FileUnload() to free a `DELIFile`. Note that you can also free the primary file with this function (for example after converting/decompressing it in InitModule() or InitSong().
Returns: `void`
Parameters:

`DELIFile file`	The file to free

FileGetData

Call FileGetData () to retrieve the contents of a `DELIFile`.
Returns:

`DELIResult`	Success or failure

Parameters:

`DELIFile file`	The file to free
`void** pData`	Pointer to pointer (!) into which DeliPlayer copies the pointer to the file data array.
`unsigned int* pSize`	Pointer to an unsigned int (!) into which DeliPlayer copies the size in bytes of the file data array.

FileGetName

Call FileGetName() to retrieve the filename of a `DELIFile`.

Returns:

`DELIResult` Success or failure.

Parameters:

`DELIFile file`	The file to free
`char* namebuffer`	A pointer to a char-array into which DeliPlayer copies the file name.
`int namebufferlen`	The size (in chars) of the char-array.

Appendix

Terminology

The following terminology explains how the terms are used in the DeliAPI 2 documentation. Not all definitions are globally valid.

Module
A set of files which belong together. To use a module, all its files must be available. Most of the time one single file builds a module, like .mp3, .mod, .it. Examples of formats with modules that consist of at least two files are TFMX and SoundTracker-Songs.

Sub-song
Modules usually contain one song (e.g. MP3s). However, certain formats (e.g. TFMX) support several songs per module. If it is important to address a specific song in a module, it is called ‘sub-song’. Most of the time it is followed by the index of the sub-song.
Example: Player ‘xyz’ plays sub-song 3 of module ‘test sound’.

One-shot part
The part of a sample based instrument that is played only once.

Loop part
The part of a sample based instrument that is repeated.

Replay state block
See the Replay State Block description.

Mixing
Mixing is the operation that combines two or more channels into one channel.

Streaming player
A streaming player is a player that does not use DeliPlayer’s mixing functionality.
Examples: WAV-player, MP3-player, OGG Vorbis–player.
Streaming players should set the `timerbase` and `timerspeed` arguments in their call to `InitReplay()` to `0`.

Tag-array
An array of {tag,value} pairs. Used in DeliPlayer for exchanging information between plugins and the core. See also DeliTag.

Interface
The definition of a set of functions to be implemented and later instantiated in one object. (in C++: a class without any members containing only virtual methods that are pure).

Class
The implementation of an interface.

Object
The instance of a class.

License

Diagrams

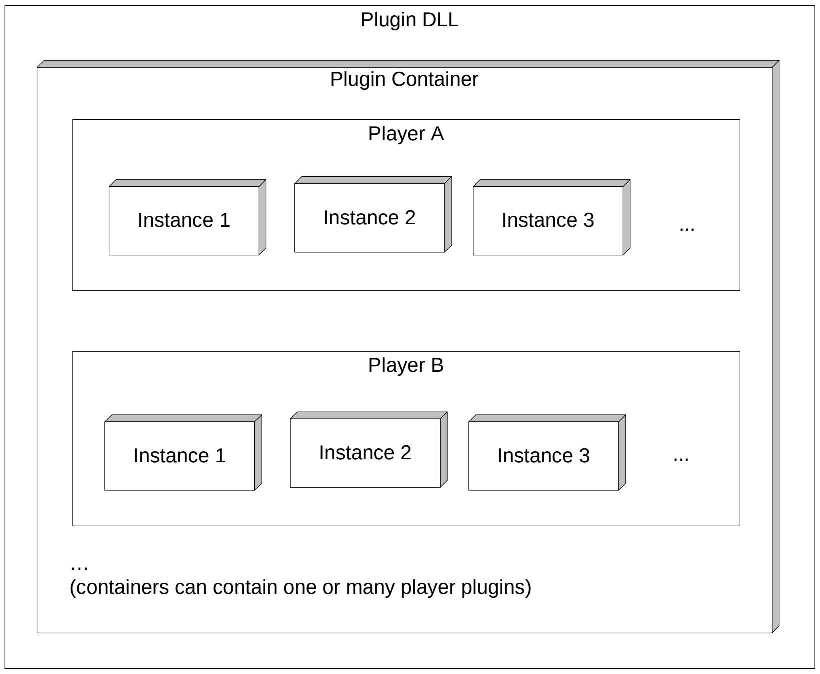


Diagram 1: Basic object structure

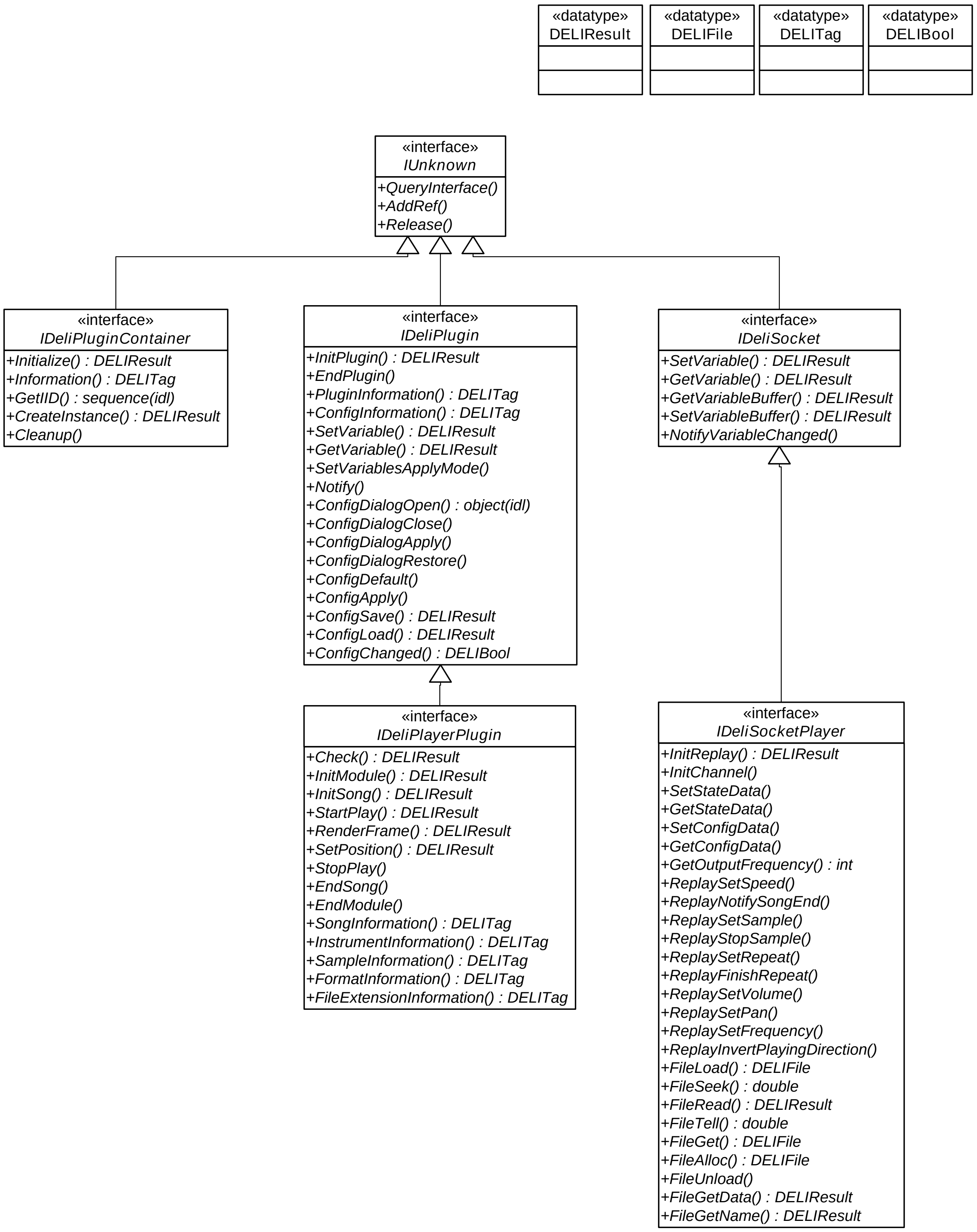


Diagram 2: DeliAPI 2 interface structure