iTunes Visual Plug-ins

Apple Applications > iTunes

2011-07-19

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Figures and Tables

iTunes Visual Plug-ins

This note is directed at application developers who want to create visual plugins for iTunes version 10.4 on Mac OS X 10.5.8 and later, and iTunes version 10.4 on Windows XP, Vista, & 7.

If you received this documentation as part of the iTunes Visual plugins SDK package, you should check the Apple Development Kits web page for an updated version.

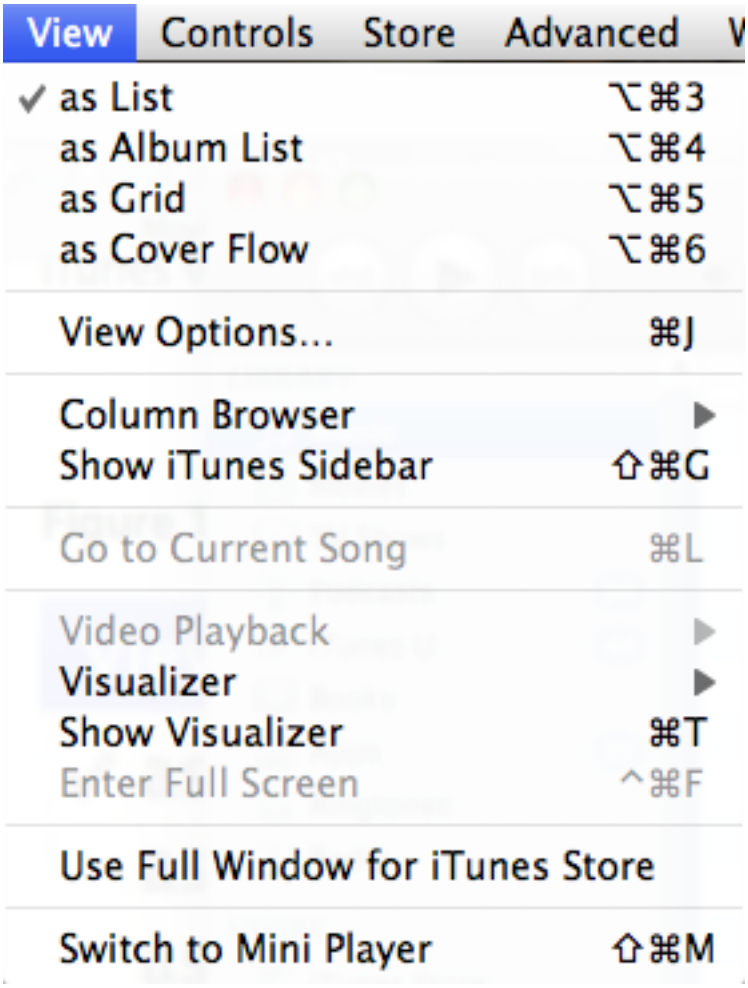
All of the code described in this documentation can also be found in the iTunes Visual plugins SDK, which is available to all registered ADC members (visit the Apple Developer Connection Membership page to learn how to become an ADC member).

We'd like to hear about your finished Visual plugin. Send information about it, and the Visual plugin itself if you like, to itunesvisuals@mac.com.

Overview

When the user selects the Show Visualizer menu item in the View menu in iTunes, custom visual special effects appear.

Figure 1-1 Turn On Visualizer menu item



iTunes plugins are implemented on Mac OS X as shared libraries, and on Windows as Dynamic Link Libraries (DLLs). For ease of implementation across different architectures, iTunes plugins export a single entry point. In addition, plugins do not link against the iTunes application. Instead, a callback function pointer is provided to your plugin during initialization, and the iTunes API calls are implemented via that function.

iTunes sends messages to your plugin to tell it when to initialize and clean up, when the user turns visual effects on and off, when the window is resized, when the user starts or stops playing music, etc. When visual effects are on, iTunes will set the focus on the platform view containing the visual effects so that your plugin can accept keyboard and mouse events. Messages are described in more detail below.

While the visual is displayed and iTunes is playing music, iTunes sends "pulse" messages that include waveform data (sound samples) corresponding to the music that is currently playing, and a spectrum analysis of the samples. During registration, your plugin indicates the number of channels of waveform and spectrum data it wants, and how often it wants to receive pulse messages. When iTunes is not playing, pulse messages are still sent but there will be no waveform or spectrum data.

The plugin can request that pulse messages be sent at a rate faster than the monitor refresh rate (typically 60Hz for LCD panels) so it can perform any calculations it needs to ensure a smooth display. However, there is no reason to draw faster than the refresh rate so there is also a separate "draw" message that is sent when

the OS determines that it is time to draw the view area covered by the visualizer. The plugin should update its internal state with every pulse message but only draw when it gets the draw message. On Mac OS X, if the plugin chooses to use an NSView-based subview, it can control the invalidation of that view and implement its drawing per the standard NSView drawing mechanism. See below for more information on this option.

During registration, your plugin can also indicate that it wants to receive idle messages. Idle messages are sent periodically whether music is playing or not. These messages are not to be used for drawing but can be used for bookkeeping.

Your plugin can have preferences that get stored in the iTunes preferences file. The iTunes API includes functions to access your plugin's preferences. iTunes sends the configure message when the user clicks on the options button to tell you to show your settings dialog.

Plug-in discovery and registration

On Mac OS X, iTunes recursively scans the plugins folder which is located at ~/Library/iTunes/iTunes Plug-ins/ (and /Library/iTunes/iTunes Plug-ins/ if that directory exists), and only that folder, for plugins. On Mac OS X, iTunes recognizes plugins packaged as bundles with `CFBundlePackageType` 'hvpl'. We recommend using iTunes' creator "hook" as the file or bundle signature so it gets an icon and kind string consistent with the other iTunes plugins.

On Windows, iTunes recursively scans the folder named "Plug-Ins", which is located beside the iTunes.exe application itself, for visual plugin files having the .dll extension.

Visual plugin sample code is provided for both Windows and Mac OS X formats in the iTunes Visual Plug-in SDK.

A plugin library exports a single entry point. The name of the exported function depends on how the plugin is packaged.

For Mac OS X plugin shared libraries, the name of the entry point must be "iTunesPluginMainMachO".

For Windows plugin DLLs, the name of the main entry point must be "iTunesPluginMain".

For each plugin found, iTunes sends a `kPluginInitMessage` message to the entry point mentioned above, which is referred to as the plugin "main" entry point in this SDK. A single plugin file can contain multiple plugins. To support this, the plugin main entry point must register each of the contained plugins when it receives the `kPluginInitMessage`, which will be sent only once.

To register a visual plugin, you call `PlayerRegisterVisualPlugin`, passing a pointer to your visual plugin message handler function. All `kVisualPlugin...` messages are sent to this handler function, starting with `kVisualPluginInitMessage`, which is sent immediately upon registration.

It is not necessary to unregister visual plugins. When it's time to shut down, a `kVisualPluginCleanupMessage` will be sent to your visual plugin message handler, and then a `kPluginCleanupMessage` will be sent your plugin's main entry point.

For compatibility with future versions of iTunes, if your plugin receives a message it does not recognize, it should return `unimpErr`.

Subviews and User Input Events on Mac OS X

On Mac OS X, visual plugins have the option of creating an NSView-based subview in which they will do their drawing. When the plugin is activated, it can create this view and add it as a subview of the view supplied by iTunes. By doing so, the plugin can gain additional control over when it draws by invalidating its view as needed instead of relying upon iTunes to do it. This will let it optimize drawing by only invalidating the areas that need to be updated instead of the whole view which is all iTunes can know about.

Furthermore, using a subview allows the plugin to get user events such as keyboard input and mouse clicks. When a visual plugin that uses a subview is active, iTunes makes sure that the subview is the window's first responder so that it gets the user events first.

During registration, the plugin should set the `kVisualUsesSubview` option bit. See the description of the `PlayerRegisterVisualPlugin` callback API below for more information. The sample code uses this option.

User Input Events on Windows

On Windows, visual plugins can get user events such as keyboard input and mouse clicks by subclassing the WndProc of the HWND supplied by iTunes when the plugin is activated. The sample code shows this technique.

The plugin's custom WndProc should not respond to the WM_PAINT message but should let iTunes handle it. Do all of your drawing in the draw message handler.

Messages in Detail

1. Plug-in main entry point messages

These messages are sent to your plugin main entry point, which has the following signature:

```
OSStatus main(OSType message, PluginMessageInfo * messageInfo, void * refCon);
```

`message`

What message is being sent

`messageInfo`

A pointer to additional parameters, if any.

`refCon`

The value you returned in `PluginInitMessage.refCon` will be passed back in this parameter.

This table lists the plugin main entry point messages and corresponding `PluginMessageInfo` variants. Messages that have no additional parameters are listed as "n/a"; you should ignore `messageInfo` for those messages.

Table 1-1 Plug-in main entry point messages

message	messageInfo->u
kPluginInitMessage	PluginInitMessage
kPluginIdleMessage	n/a
kPluginPrepareToQuitMessage	n/a
kPluginCleanupMessage	n/a

kPluginInitMessage

This message is sent to your plugin library at launch time. You should register your plugins by calling `PlayerRegisterVisualPlugin`, described below, when you get this message. Then fill in the `options` and `refCon` fields of the `PluginInitMessage` before returning.

`messageInfo->u.initMessage` is a `PluginInitMessage`:

```
enum {
    /* PluginInitMessage.options */

    /* Send idle messages to plugin main */
    kPluginWantsIdleMessages        = (1u << 1),

   /* Don't close this plugin just because it didn't register anyone right away
 */
    kPluginWantsToBeLeftOpen        = (1u << 2),

    /* The plugin wants to know when the display depth/size changes */
    kPluginWantsDisplayNotification = (1u << 5)
};

struct PluginInitMessage {
 UInt32       majorVersion; /* Input */
 UInt32       minorVersion; /* Input */

 void *       appCookie; /* Input */
 ITAppProcPtr appProc; /* Input */

 OptionBits   options; /* Output */
 void *       refCon; /* Output */
};
```

`majorVersion minorVersion`

The version of the iTunes API implemented by the iTunes application.

`appCookie appProc`

Parameters to be passed to the callback APIs

`options`

Return options parameters (see flags above) to specify whether you want to receive idle messages, know about display depth/size changes, and so on.

`refCon`

The value you return here will be passed back as the `refCon` parameter in future calls to your main entry point

kPluginIdleMessage

Visual plugins that want idle time should register for `kVisualPluginIdleMessage` instead, described below.

kPluginPrepareToQuitMessage

This message is sent to your plugin when iTunes is about to quit.

kPluginCleanupMessage

This message is sent to your plugin when your plugin is being unloaded as iTunes quits.

2. Visual plugin messages

A visual plugin message handler has the following signature:

```
OSStatus VisualPluginHandler(OSType message, VisualPluginMessageInfo *
messageInfo, void * refCon);
```

`message`

What message is being sent.

`messageInfo`

A pointer to additional parameters, if any.

`refCon`

The value you returned in `VisualPluginInitMessage.refCon` will be passed back in this parameter.

This table lists the visual plugin messages and corresponding `VisualPluginMessageInfo` variants. Messages that have no additional parameters are listed as "n/a", you should ignore `messageInfo` for those messages.

Table 1-2 Visual plugin messages

message	messageInfo->u
kVisualPluginInitMessage	VisualPluginInitMessage
kVisualPluginCleanupMessage	n/a

message	messageInfo->u
kVisualPluginIdleMessage	n/a
kVisualPluginConfigureMessage	n/a
kVisualPluginEnableMessage	n/a
kVisualPluginDisableMessage	n/a
kVisualPluginActivateMessage	VisualPluginActivateMessage
kVisualPluginDeactivateMessage	n/a
kVisualPluginWindowChangedMessage	VisualPluginWindowChangedMessage
kVisualPluginPulseMessage	VisualPluginPulseMessage
kVisualPluginDrawMessage	n/a
kVisualPluginFrameChangedMessage	n/a
kVisualPluginPlayMessage	VisualPluginPlayMessage
kVisualPluginChangeTrackMessage	VisualPluginChangeTrackMessage
kVisualPluginStopMessage	n/a
kVisualPluginSetPositionMessage	VisualPluginSetPositionMessage
kVisualPluginCovertArtMessage	VisualPluginCoverArtMessage
kVisualPluginDisplayChangedMessage	VisualPluginDisplayChangedMessage

kVisualPluginInitMessage

This message is sent right after you register your plugin with `PlayerRegisterVisualPlugin`. For this message only, the `refCon` parameter will be the value you returned in `PlayerRegisterVisualPluginMessage.registerRefCon`. Fill in the `refCon` fields of the `VisualPluginInitMessage` before returning.

`messageInfo->u.visualPluginInitMessage` is a `VisualPluginInitMessage`:

```
struct VisualPluginInitMessage {
 UInt32       messageMajorVersion; /* Input */
 UInt32       messageMinorVersion; /* Input */
 NumVersion   appVersion; /* Input */

 void *       appCookie; /* Input */
 ITAppProcPtr appProc; /* Input */

 OptionBits   unused; /* N/A */
 void *       refCon; /* Output */
};
```

`messageMajorVersion messageMinorVersion`

The version of the iTunes API implemented by the iTunes application.

`appVersion`

The version of iTunes that is running.

`appCookie appProc`

The plugin should copy these two fields into its private data since it will need to pass them as parameters to all the iTunes APIs described below.

`unused`

Ignore. Set to 0.

`refCon`

The value returned in this field will be passed as the `refCon` parameter in subsequent calls to the visual plugin handler. Your plugin can use this for anything it chooses, typically to store a pointer to data which is allocated in the init message. The sample code uses this technique to recover the pointer to its private data in its visual plugin handler.

kVisualPluginCleanupMessage

This message is sent when iTunes is about to quit. You should free any resources allocated by your visual plugin at this time.

kVisualPluginIdleMessage

This message is sent periodically if the plugin requests idle messages. Do this by setting the `kVisualWantsIdleMessages` option in the `PlayerRegisterVisualPluginMessage.options` field. Your plugin should NOT draw at this time, it is for internal book-keeping only.

kVisualPluginConfigureMessage

This message is sent when the user picks the Options... item from the Visualizer sub-menu of the View menu. Enable the Options... menu item (and this message) by setting the `kVisualWantsConfigure` option in the `PlayerRegisterVisualPluginMessage.options` field. The sample code shows how to implement a settings dialog for configuration.

kVisualPluginEnableMessage

kVisualPluginDisableMessage

iTunes currently enables all loaded visual plugins. Your plugin should simply return `noErr` for these messages.

kVisualPluginActivateMessage

Sent when visual effects are turned on. At this point, the plugin should allocate any large buffers it needs.

`messageInfo->u.activateMessage` is a `VisualPluginActivateMessage`:

```
struct VisualPluginActivateMessage {     VISUAL_PLATFORM_VIEW            view;
 /* Input */     OptionBits                      options; /* Input */ };
```

`view`

The platform "view" to draw into. The plugin should remember this since it is not sent with any other messages. On Mac OS X, this is an NSView*. On Windows, it is a child HWND of the iTunes window.

`options`

The only option currently defined is `kWindowIsFullScreen`, it's set when the visuals are being displayed in full screen mode.

kVisualPluginWindowChangedMessage

This message is sent when the visualizer is moved to a new window (e.g. from a normal window to fullscreen).

`messageInfo->u.windowChangedMessage` is a `VisualPluginWindowMessage`:

```
struct VisualPluginWindowChangedMessage {     OptionBits
options; /* Input */ };
```

`options`

The only option currently defined is `kWindowIsFullScreen`, it's set when the visuals are being displayed in full screen mode.

kVisualPluginDeactivateMessage

This message is sent when visual effects are turned off. Your plugin should free any large buffers allocated in the course of rendering here. It should also forget about the VISUAL_PLATFORM_VIEW previously sent in the activate message as it will no longer exist after you return from handling this message.

kVisualPluginFrameChangedMessage

This message is sent when the user resizes the iTunes window. Your plugin should adjust its drawing rectangle accordingly based on the current frame value of the VISUAL_PLATFORM_VIEW previously sent in the activate message.

kVisualPluginPulseMessage

This message is sent periodically when the visualizer is active, at a rate specified during registration. The intent of this message is to supply data fast enough for your plugin to perform calculations on the data and update its internal drawing state. While it is safe to draw at this time, it is not recommended. It is preferred that you wait until the draw message is sent before doing any drawing.

`messageInfo->u.pulseMessage` is a `VisualPluginPulseMessage`:

```
struct VisualPluginPulseMessage {
    RenderVisualData *              renderData; /* Input */
    UInt32                          timeStampID; /* Input */
    UInt32                          currentPositionInMS; /* Input */

    UInt32                          newPulseRateInHz; /* Input/Output - contains
 current rate on input, modify it to get a new rate. */
};

struct RenderVisualData {
 UInt8 numWaveformChannels;
 UInt8 waveformData[kVisualMaxDataChannels][kVisualNumWaveformEntries];

 UInt8 numSpectrumChannels;
 UInt8 spectrumData[kVisualMaxDataChannels][kVisualNumSpectrumEntries];
};
```

`renderData->numWaveformChannels`

The number of channels of waveform data included. This will be the number you requested during registration.

`renderData->waveformData`

The most significant 8 bits of the sound samples that are currently playing. The values range from 0 to 255, where 128 is the midpoint (AC zero value).

`renderData->numSpectrumChannels`

The number of channels of spectrum data included. This will be the number you requested during registration.

`renderData->spectrumData`

This is a 512-point Fast Fourier Transform of the waveform data.

kVisualPluginDrawMessage

This message is sent when it is time for the plugin to draw. This message is synchronized with the OS and is sent when the OS decides it is time to redraw the region of the window covered by the visuals view.

kVisualPluginPlayMessage

This message is sent when iTunes starts playing a track. Your plugin should copy any track info it wants to display. Also, when streaming an internet radio station the track info contains the name (and meta data) for the radio station, and the song name (and its meta data) is in the `ITStreamInfo` field. Your plugin should not draw at this time, a draw message will be sent as soon as possible.

`messageInfo->u.playMessage` is a `VisualPluginPlayMessage`:

```
struct VisualPluginPlayMessage {
    ITTrackInfo *                   trackInfo;   /* Input */
    ITStreamInfo *                  streamInfo;  /* Input */

    AudioStreamBasicDescription     audioFormat; /* Input */
```

```
    UInt32                          bitRate;     /* Input */
    SInt32                          volume;      /* Input */
};
```

kVisualPluginChangeTrackMessage

This message is sent when the information about a track changes, e.g., when the user edits track info, or when iTunes begins playing a different track. Your plugin should copy any track info it wants to display but should not draw at this time, a draw message will be sent as soon as possible. Also, when streaming an internet radio station the track info contains the name (and meta data) for the radio station, and the song name (and its meta data) is in the `ITStreamInfo` field.

`messageInfo->u.changeTrackMessage` is a `VisualPluginChangeTrackMessage`:

```
struct VisualPluginChangeTrackMessage {     ITTrackInfo *
trackInfo;  /* Input */     ITStreamInfo *                streamInfo; /* Input
 */ };
```

kVisualPluginStopMessage

This message is sent when the music stops playing.

kVisualPluginSetPositionMessage

This message is sent when iTunes changes the elapsed time position within a track. A plugin that shows the elapsed time would use this.

`messageInfo->u.setPositionMessage` is a `VisualPluginSetPositionMessage`:

```
struct VisualPluginSetPositionMessage {     UInt32
positionTimeInMS; /* Input */ };
```

kVisualPluginCoverArtMessage

This message is sent in response to a request for the currently playing track's cover art.

`messageInfo->u.coverArtMessage` is a `VisualPluginCoverArtMessage`:

```
struct VisualPluginCoverArtMessage {
    VISUAL_PLATFORM_DATA    coverArt;       /* Input - platform-specific artwork
 data representation */
    UInt32                  coverArtSize;   /* Input - size of the coverArt in
 bytes */
    UInt32                  coverArtFormat; /* Input - format of cover art */
};
```

`coverArt`

A platform-specific pointer to the cover art data. On the Mac, this is a CFDataRef. On Windows, it is a LPCVOID pointer. The data is disposed after this call returns so the plugin must not save off the pointer and expect it to be valid later. If the plugin wants to keep the data around for later, it needs to retain it (Mac) or copy it (Windows). If the currently playing track has no artwork, this pointer will be nil.

`coverArtSize`

The size of the artwork image in bytes.

`coverArtFormat`

An enumeration that specifies what type of artwork image data is represented.

```
enum {     /* CoverArt format types */     kVisualCoverArtFormatJPEG   = 13,
   kVisualCoverArtFormatPNG    = 14,     kVisualCoverArtFormatBMP    = 27 };
```

kVisualPluginDisplayChangedMessage

This message is sent when something about the display environment (bit depth, for example) has changed. If your plugin handles the event it should return `noErr`. Otherwise it should return `unimpErr`

`messageInfo->u.displayChangedMessage` is a `VisualPluginDisplayChangedMessage` :

```
enum {
    kVisualDisplayDepthChanged  = 1 << 0, /* the display's depth has changed */

    kVisualDisplayRectChanged   = 1 << 1, /* the display's location changed */
    kVisualWindowMovedMoved     = 1 << 2, /* the window has moved location */
    kVisualDisplayConfigChanged = 1 << 3, /* something else about the display
changed */
};

struct VisualPluginDisplayChangedMessage {
    UInt32       flags; /* Input */
};
```

Callback APIs

As mentioned in the overview, plugins do not link against iTunes. Instead, the first two parameters of all of the iTunes APIs are a cookie and a callback function pointer. These are provided to your plugin in the init messages. The glue code in iTunesAPI.c takes care of marshaling parameters and calling iTunes for you, so you don't have to worry about the details.

```
OSStatus PlayerRegisterVisualPlugin(void *appCookie, ITAppProcPtr appProc,
PlayerMessageInfo *messageInfo);
```

Register your visual plugin with `PlayerRegisterVisualPlugin` when you receive `kPluginInitMessage`. Pass a `PlayerRegisterVisualPluginMessage` variant of `PlayerMessageInfo` as the `messageInfo` parameter. The fields are explained in the comments below:

```
 /* PlayerRegisterVisualPluginMessage.options */

enum {
    /* Plugin uses 3D exclusively (OpenGL or Direct3D), iTunes does not need to
 invalidate the backing native view. */
    kVisualUsesOnly3D                   = (1u << 0),

    /* Plugin supports discrete or integrated graphics without requiring a mode
```

```
 switch. */
    kVisualSupportsMuxedGraphics        = (1u << 1),

    /* Mac-only: Plugin adds an NSView-based subview to the one provided, iTunes
 does not need to invalidate the backing native view.*/
    kVisualUsesSubview                  = (1u << 2),

    /* Plugin wants periodic idle messages.  Note that these idle messages are
 not for drawing, only for internal processing. */
    kVisualWantsIdleMessages            = (1u << 3),

    /* Plugin supports a configure dialog, iTunes will display an item in the
menus to show it. */
    kVisualWantsConfigure               = (1u << 5)
};

struct PlayerRegisterVisualPluginMessage {
    /* Input from plugin */

    ITUniStr255                    name;          /* Displayed in the Visualizer
 menu */
    OptionBits                      options;        /* See above */
    OSType                         creator;        /* Identifies the plugin */

    NumVersion                      pluginVersion;  /* Version number of the
plugin */

    VisualPluginProcPtr            handler;        /* Handler for the plugin's
 messages */
    void *                          registerRefCon; /* RefCon for the plugin's
 handler */

    UInt32                          pulseRateInHz;       /* Send the plugin a
"pulse" message N times a second (max ~= 120 but may vary) */
    UInt32                          numWaveformChannels; /* 0-2 waveforms
requested */
    UInt32                          numSpectrumChannels; /* 0-2 spectrums
requested */

    SInt32                          minWidth;  /* Minimum resizeable width (0
for no restriction) */
    SInt32                          minHeight; /* Minimum resizeable height (0
 for no restriction) */

    SInt32                          maxWidth;  /* Maximum resizeable width (0
for no limit) */
    SInt32                          maxHeight; /* Maximum resizeable height (0
 for no limit) */
};
```

Show the "About iTunes" window.

```
void PlayerShowAbout(     void *appCookie,     ITAppProcPtr appProc);
```

Tell iTunes to open a URL. The `urlString` parameter is not a Pascal string or a C string; its length is specified by the `length` parameter.

```
void PlayerOpenURL (     void *appCookie,     ITAppProcPtr appProc,     SInt8
* urlString,     UInt32 length);
```

Read data identified by your plugin's name from the iTunes preferences file. This can be used to store your plugin's preferences as a single block of data. Your plugin's name is specified in `PlayerRegisterVisualPluginMessage.name`.

```
OSStatus PlayerGetPluginData (     void *appCookie,     ITAppProcPtr appProc,
    void *dataPtr,     UInt32 dataBufferSize,     UInt32 *dataSize);
```

Save data identified by your plugin's name to the iTunes preferences file. This can be used to store your plugin's preferences as a single block of data. Your plugin's name is specified in `PlayerRegisterVisualPluginMessage.name`.

```
OSStatus PlayerSetPluginData (     void *appCookie,     ITAppProcPtr appProc,
    void *dataPtr,     UInt32 dataSize);
```

Read data identified by `dataName` and your plugin's name from the iTunes preferences file. This can be used to store your plugin's preferences, with each preference field stored separately. Your plugin's name is specified in `PlayerRegisterVisualPluginMessage.name`.

```
OSStatus PlayerGetPluginNamedData (     void *appCookie,     ITAppProcPtr appProc,
     ConstStringPtr dataName,     void *dataPtr,     UInt32 dataBufferSize,
  UInt32 *dataSize);
```

Save data identified by `dataName` and your plugin's name to the iTunes preferences file. This can be used to store your plugin's preferences, with each preference field stored separately. Your plugin's name is specified in `PlayerRegisterVisualPluginMessage.name`.

```
OSStatus PlayerSetPluginNamedData (     void *appCookie,     ITAppProcPtr appProc,
     ConstStringPtr dataName,     void *dataPtr,     UInt32 dataSize);
```

Tell iTunes to enter or exit full screen mode. If your plugin wants to behave like a screen saver it could use this.

```
OSStatus PlayerSetFullScreen (     void *appCookie,     ITAppProcPtr appProc,
    Boolean fullScreen);
```

Request the track cover art for the currently playing song. This request is asynchronous and the data will be returned later via a `kVisualPluginCoverArtMessage` call. See above for more details.

```
OSStatus PlayerRequestCurrentTrackCoverArt (     void *appCookie,
ITAppProcPtr appProc);
```

Your plugin should call `PlayerIdle` to give time to iTunes if it engages in a lengthy process. The plugin is highly discouraged from blocking iTunes for long periods of time since all visual plugin calls occur on the main thread. You should not be blocking the iTunes UI with lengthy operations.

```
OSStatus PlayerIdle (     void *appCookie,     ITAppProcPtr appProc);
```

Document Revision History

This table describes the changes to *iTunes Visual Plug-ins*.

Date	Notes
2011-07-18	Update to iTunes version 10.4 interfaces
2007-11-13	Update to iTunes version 7.4 interfaces
2001-06-26	New document that describes the APIs used by iTunes and its Visual plug-ins to communicate back and forth.

Document Revision History