Gig Performer® 4 SDK Version 4.9
Deskew Technologies, LLC Confidential
February 28th, 2023

Prolog

Before you consider writing a library for use with Gig Performer, please familiarize yourself with the operation of Gig Performer itself so that you have a reasonable understanding of its capabilities and functionality that might impact your library.

Introduction

Starting with Gig Performer 4, Gig Performer (GP from now on) exposes an API that allows third party developers to create libraries that can be used to control GP and to respond to events in GP. The primary purpose of the API is to enable the creation of libraries to manage control surfaces and other hardware that one might want to use with GP. This SDK includes the source code that will need to be included in every third party project as well as some samples that will help to get started.

Fundamentals

1) GP will query all libraries found in /Users/Shared/Gig Performer/ThirdParty (OS X) and in C:\Users\All Users\Gig Performer\ThirdParty (Windows) to collection information (product name, build date, description, etc). This information will be displayed to the user who can decide which libraries should be loaded and used.
2) GP will next load the desired libraries.
3) GP exposes a collection of functions that a library can call to interact with the application. For example, you can set the current value of a widget[1] in a rackspace. You can ask for the number of songs in the current setlist, get the name of each song, get the names of all the song parts in a song, switch from one song to another and so forth. The library can of course also send MIDI events into GP.
4) The GP API supports callbacks into the third party library to notify the library that something "happened". For example, if the user switches from one song to another, a library supporting a keyboard controller with a programmable display could be notified so that it can display the names of the song parts in the new song. If the library is interested in MIDI events coming from a particular piece of hardware, it can be called whenever that hardware sends out MIDI data.
5) The GP API has a mechanism that allows developers to define GP Script functions that can be injected into Gig Performer to invoke code implemented with the extension.

[1] The widget must have a handle defined

Structure of the API

The underlying API is defined as a collection of typedefs and functions in C as this will allow almost any language to be used to develop a library as long as it can make C calls. Simple examples are provided in C++ and shell scripts are included to automate the build process. The example class included for the C++ example will perform some initial operations automatically so that the user does not need to be concerned with low level setup.

Working with the API

When GP is first loaded, it will looked for third party libraries. For each library that it finds, the following sequence of steps will be performed

1) GP will open the library
2) GP will look for a function called GPQueryLibrary. If found, it will be invoked so that the library can return useful information about itself thereby allowing the user to decide whether to use a particular library. The information should be returned as a small XML formatted string.
   ```
   extern "C" EXPORTED void GPQueryLibrary(char* xmlInfoBuffer, int bufferLength);
   ```
3) GP will then close the library
4) For all libraries that the user chooses to load, GP will then open the library again and will look for a function called GPRegister
   ```
   const char* GPRegister(TGetGPFunctionType getGPFunctionAddress, void* handle)
   ```
5) If the function is found, it will be called with two parameters:
   a. `void* (*TGetGPFunctionType)(void* handle, const char* functionName)` The first parameter is the address of a function that the library can use to call back into GP to request the address of each function that GP exposes to the API.
   b. The second parameter, `void* handle`, is an opaque handle whose value should be stored and included as a parameter in all subsequent calls into Gig Performer.
   c. The function should return a string that represents the name of the control surface or whatever product this library supports. That name will show up in a list that the user can view and choose to enable or disable.
6) Your implementation of this function should include calls to get the addresses of all the other available functions exposed by GP so that you can use them later. Note that the C++ wrapper will do this automatically. Here is an example of the raw GPRegister function in raw C along with a single callback

```
extern “C” EXPORTED void GPRegister(TGetGPFunctionType getGPFunctionAddress,
void* handle)
    {
       Handle = handle; Store the handle for future use
       InitializeImportedFunctions(handle, getGPFunctionAddress);

       //You will need to register callbacks to be informed of changes
```

```
        GP_RegisterCallback(handle, "OnSongChanged");
    }

extern “C” EXPORTED void OnSongChanged(int oldIndex, int newIndex)
    {
        char str[255];
        sprintf(str, "Song changed from: %d, to: %d", oldIndex, newIndex);
        GP_Log(Handle, str);
    }
```

7) Register for whatever other callbacks your library will need. However, to keep your library mean and lean, please don’t just arbitrarily register for all callbacks.
8) Your library is now ready to interact with Gig Performer and whatever hardware you are controlling

Function list

Following is a list of C functions exported by GP that can be called from a library

---

bool GP_RegisterCallback(LibraryHandle h, const char* callbackName);
Specify the name of the callback function to be called when the respective event occurs. The callback must be implemented in your library as an exported function with a name known to GP

---

bool GP_UnregisterCallback(LibraryHandle h, const char* callbackName);
Inform GP that you no longer want to be called back when the respective event occurs.

---

void GP_SetPlayheadState(LibraryHandle h, bool play);
Enable or disable the global playhead.

---

bool GP_GetPlayheadState(LibraryHandle h);
Get the global playhead state

---

void GP_ShowTuner(LibraryHandle h, bool show);
Switch into or out of tuner view.

---

bool GP_TunerShown(LibraryHandle h);
Indicates whether the tuner is visible.

---

void GP_EnableMetronome(LibraryHandle h, bool enable);
Turn the metronome on or off.

---

bool GP_MetronomeEnabled(LibraryHandle h);
Indicates whether the metronome is turned on.

---

int GP_GetPathToMe(LibraryHandle h, char* returnBuffer, int bufferLength);
Returns the folder path where your library is installed. It is useful when you need to access other resource files installed in the same or a relative location to the library.

---

int GP_GetPathToMe(LibraryHandle h, char* returnBuffer, int bufferLength);
Returns the folder path where your library is installed. It is useful when you need to access other resource files installed in the same or a relative location to the library.

---

void GP_SwitchToSetlistView(LibraryHandle h);
Switch Gig Performer to Setlist View

---

void GP_SwitchToWiringView(LibraryHandle h);
Switch Gig Performer to the plugin wiring view

void GP_SwitchToPanelView(LibraryHandle h);
Switch Gig Performer to the front panel and widgets view

---

int GP_GetPluginList(LibraryHandle h, char* returnBuffer,
int bufferLength, bool useGlobalRackspace);
Get a list of all plugins defined (i.e, with handles) in the currently active rackspace or in the global rackspace. The return value is the actual number of characters needed for the return value.

---

bool GP_PluginExists(LibraryHandle h, const char* pluginHandle, bool useGlobalRackspace);
Returns whether the plugin with the given handle exists in the currently active rackspace or in the global rackspace.

---

bool GP_SetPluginParameter(LibraryHandle h, const char* pluginHandle, int parameterNumber, double value, bool useGlobalRackspace);
Set a parameter value for a plugin with the given handle in the currently active rackspace or in the global rackspace.

---

double GP_GetPluginParameter(LibraryHandle h, const char* pluginHandle, int parameterNumber, bool useGlobalRackspace);
Returns the value of the given parameter in the plugin with the given handle in the currently active rackspace or in the global rackspace.

---

double GP_GetPluginParameterCount(LibraryHandle h, const char* pluginHandle, bool useGlobalRackspace);
Returns the number of parameters exposed by the plugin with the given handle in the currently active rackspace or in the global rackspace.

---

int GP_GetPluginParameterName(LibraryHandle h, const char* pluginHandle, int parameterNumber, char* returnBuffer, int bufferLength, bool useGlobalRackspace);
Returns the name of the parameter at the specified parameter number of the plugin with the given handle in the currently active rackspace or in the global rackspace.

---

int GP_GetPluginParameterText(LibraryHandle h, const char* pluginHandle, int parameterNumber, char* returnBuffer, int bufferLength, bool useGlobalRackspace);
Returns the text value of the parameter at the specified parameter number of the plugin with the given handle in the currently active rackspace or in the global rackspace.

```
int GP_GetWidgetList(LibraryHandle h, char* returnBuffer,
                     int bufferLength, bool useGlobalRackspace);
```

Get a list of all widgets defined (i.e, with handles) in the currently active rackspace or in the global rackspace. The return value is the actual number of characters needed for the return value.

```
bool GP_WidgetExists(LibraryHandle h, const char*  widgetName);
```

Query GP to see if a widget with the given name exists in the currently active rackspace

---

```
double GP_GetWidgetValue(LibraryHandle h, const char*  widgetName);
```

Get the current value (widget position) of the named widget

```
int GP_GetWidgetTextValue(LibraryHandle h, const char*  widgetName,
                          char* returnBuffer, int bufferLength);
```

Get the current text value of the named widget. The text value may not be numeric if the parameter connected to this widget provides one. The return value is the actual number of characters needed for the return value.

```
bool GP_SetWidgetValue(LibraryHandle h, const char*  widgetName,
                       double newValue);
```

Set the value of a named widget.

```
bool GP_ResetWidgetToDefault(LibraryHandle h, const char*  widgetName,
                             double newDefault);
```

Resets a widget back to its predefined default value if newDefault is -1. If newDefault is between 0.0 and 1.0 then save this as new default value but do not change the widget value

---

```
bool GP_SetWidgetCaption(LibraryHandle h, const char*  widgetName,
                         const char*  newCaption);
```

Set the label for a widget

```
int GP_GetWidgetCaption(LibraryHandle h, CharPtr widgetName,
                        char* returnBuffer, int bufferLength)
```

Get the label for a widget. The library must provide a character buffer and indicate its size. GP will fill in the buffer with the caption. The return value is the actual number of characters needed for the return value.

```
void GP_SetWidgetHideOnPresentation)(LibraryHandle h, const char*
widgetName, bool hide);
```

Set whether widget should be displayed when not editing

```
void GP_SetWidgetBounds)(LibraryHandle h, const char*  widgetName, int left,
int top, int width, int height);
```

Set the position and size of the widget

void GP_GetWidgetBounds)(LibraryHandle h, const char* widgetName, int * left, int * top, int * width, int * height);
Get the position and size of the widget

---

bool GP_GetWidgetHideState)(LibraryHandle h, const char* widgetName);
Get whether the widget is visible when not editing

---

bool GP_ListenForWidget(LibraryHandle h, const char* widgetName, bool listen);
Tell GP to call (or stop calling) the OnWidgetValueChanged callback for the named widget

---

int GP_RGBAToColor(LibraryHandle h, double red, double green, double blue, double alpha);
Convert RGB+Alpha values (all between 0.0 and 1.0) into a result that can be used to set colors where available

---

void GP_ColorToRGBA(LibraryHandle h, int color, double* red, double* green, double* blue, double* alpha);
Convert an integer color back into RGB+Alpha values (each between 0.0 and 1.0)

---

void GP_SetWidgetFillColor(LibraryHandle h, const char* widgetName, int color);
Set the fill color of widgets that support it

---

int GP_GetWidgetFillColor(LibraryHandle h, const char* widgetName);
Get the fill color of widgets that support it

---

void GP_SetWidgetOutlineColor(LibraryHandle h, const char* widgetName, int color);
Set the outline color of widgets that support it

---

int GP_GetWidgetOutlineColor(LibraryHandle h, const char* widgetName);
Get the outline color of widgets that support it

---

void GP_SetWidgetOutlineThickness(LibraryHandle h, const char* widgetName, int thickness);
Set the outline thinkness of widgets that support it

---

int GP_GetWidgetOutlineThickness(LibraryHandle h, const char* widgetName);
Get the outline thinkness of widgets that support it

---

void GP_SetWidgetOutlineRoundness(LibraryHandle h, const char* widgetName, int roundness);
Set the outline roundness of widgets that support it

---

int GP_GetWidgetOutlineRoundness(LibraryHandle h, const char* widgetName);
Get the outline roundness of widgets that support it

---

bool GP_ListeningForWidget(LibraryHandle h, const char* widgetName);
Query GP to see if you are listening for widget changes

---

bool GP_ListenForMidi(LibraryHandle h, const char* deviceName, bool listen);
Tell GP to call (or stop calling) the OnMidiIn callback when MIDI events arrive at the specified device name

---

bool GP_ListeningForMidi(LibraryHandle h, const char* deviceName);
Query GP to see if you are listening for MIDI events from a specified device name

---

int GP_GetMidiInDeviceCount(LibraryHandle h);
Query GP for the number of available Midi In devices

---

void GP_Panic(LibraryHandle h);
Send once to trigger MIDI All Notes Off. Send twice in succession to reset audio system

---

int GP_GetMidiInDeviceName(LibraryHandle h, int index,
char* returnBuffer, int bufferLength);
Query GP for the name of a Midi Input device at the specified index (zero-based). The library must provide a character buffer and indicate its size. GP will fill in the buffer with the name. The return value represents the actual length needed to fill the buffer.

---

int GP_GetMidiOutDeviceCount(LibraryHandle h);
Query GP for the number of available Midi Out devices

---

int GP_GetMidiOutDeviceName(LibraryHandle h, int index,
char* returnBuffer, int bufferLength);
Query GP for the name of a Midi Output device at the specified index (zero-based). The library must provide a character buffer and indicate its size. GP will fill in the buffer with the name. The return value represents the actual length needed to fill the buffer.

---

int (*TGP_TextToHexString)(LibraryHandle h, const char* text,
uint8_t* returnBuffer, int bufferLength);
Convert ascii text into a hex string, suitable for including in a sysex message

---

void GP_SendMidiMessageToMidiOutDevice(LibraryHandle h,
const char* deviceName,
const uint8_t* midiData, int length);
Send a MIDI message to the specified Midi Output device. It is the library's responsibility to ensure that the message is valid

```
void GP_InjectMidiMessageToMidiInputDevice(LibraryHandle h,
                                           const char* deviceName,
                                           const uint8_t* midiData,
                                           int length);
```
Insert a Midi message into Gig Performer such that GP will respond to it as if it actually came from the specified device. It is the library's responsibility to ensure that the message is valid

---

```
void GP_InjectMidiMessageToMidiInputAlias(LibraryHandle h,
                                          const char* rigManagerAlias,
                                          const uint8_t* midiData,
                                          int length);
```
Insert a Midi message into Gig Performer such that GP will respond to it as if it actually came from the specified Rig Manager global name/alias. It is the library's responsibility to ensure that the message is valid.

```
int GP_GetSongCount(LibraryHandle h);
```
Query GP for the list of songs currently available

```
int GP_GetSongName(LibraryHandle h, int atSongIndex,
                   char* returnBuffer, int bufferLength);
```
Query GP for the name of a song at the given index (zero based). The library must provide a character buffer and indicate its size. GP will fill in the buffer with the name. The return value represents the actual length needed to fill the buffer.

```
int GP_GetArtistName(LibraryHandle h, int atSongIndex,
                   char* returnBuffer, int bufferLength);
```
Query GP for the name of the artist for the song at the given index (zero based). The library must provide a character buffer and indicate its size. GP will fill in the buffer with the name. The return value represents the actual length needed to fill the buffer.

```
int GP_GetCurrentSongIndex(LibraryHandle h);
```
Query GP for the index of the currently selected song

```
int GP_GetSongpartCount(LibraryHandle h, int atSongIndex);
```
Query GP for the number of song parts in the song at the given index

---

```
int GP_GetSongpartName(LibraryHandle h, int atSongIndex, int atIndex, char*
returnBuffer, int bufferLength);
```
Query GP for the name of the song part at the specified (zero-based) index of the specified song. The library must provide a character buffer and indicate its size. GP will fill in the buffer with the name. The return value represents the actual length needed to fill the buffer.

int GP_GetVariationNameForSongPart(LibraryHandle h, int atSongIndex, int atIndex, char* returnBuffer, int bufferLength);
Query GP for the name of the underlying variation associated with the part at the specified (zero-based) index of the specified song. The library must provide a character buffer and indicate its size. GP will fill in the buffer with the name. The return value represents the actual length needed to fill the buffer.

---

int GP_GetCurrentSongpartIndex(LibraryHandle h);
Query GP for the index of the currently selected song part index

---

bool GP_InSetlistMode(LibraryHandle h);
Query GP to see if it is in setlist mode

---

bool GP_SwitchToSong(LibraryHandle h, int songIndex, int partIndex);
Tell GP to switch to the song and song part with the given indices

---

bool GP_SwitchToSongPart(LibraryHandle h, int partIndex);
Tell GP to switch to the song part with the given index in the current song

---

void GP_ConsoleLog(LibraryHandle h, const char* message);
Send the message to the console output (works only when Xcode is open)

---

void GP_ScriptLog(LibraryHandle h, const char* message, bool openLogWindow);
Display the message in the Script Logger window, possibly opening the window if it's not already open

---

void GP_GetCurrentTimeSignature(LibraryHandle h, int* numerator, int* denominator);
Returns the current time signature

---

int GP_GetRackspaceCount(LibraryHandle h);
Returns the number of rackspaces in the gig file

---

int GP_GetRackspaceName(LibraryHandle h, int atIndex,
char* returnBuffer, int bufferLength);
Get the name of rackspace atIndex

---

int GP_GetCurrentRackspaceIndex(LibraryHandle h);
Get the rackspace currently in use

---

int GP_GetCurrentVariationIndex(LibraryHandle h);
Get the currently selected variation

---

int GP_GetVariationCount(LibraryHandle h, int atRackspaceIndex);
Get the number of variations in the given rackspace

---

int GP_GetVariationName(LibraryHandle h,int atRackspaceIndex, int atIndex, char* returnBuffer, int bufferLength);
Get the name of a specific variation at the given rackspace index

---

bool GP_SwitchToRackspace(LibraryHandle h, int rackspaceIndex, int variationIndex);
Switch to the given rackspace and variation. If rackspaceIndex is -1 this will only change the variation of the active rackspace. Returns whether successful

---

bool GP_SwitchToRackspaceName(LibraryHandle h, const char* rackspaceName, const char* variationName);
Switch to the named rackspace and variation. Returns whether successful

---

int GP_GetInstanceName(LibraryHandle h, char* returnBuffer, int bufferLength);
Get the name of the currently running instance

---

bool saveGigUnconditionally(LibraryHandle h, bool withTimestamp);
Try to save the file (normally overwriting the existing one). However, if withTimestamp is true, then create a subfolder with the same name as the gigfile and save the current file in it, appending the current time to it. (Time Stamp not yet implemented)

---

bool loadGigByIndex(LibraryHandle h, int indexNumber);
Look for a gigfile in a special folder (TBD) that has the prefix indexNumber and try to load it. NOT YET IMPLEMENTED

---

void GP_Tap(LibraryHandle h);
Same as clicking the global TapTempo button

---

void GP_SetBPM(LibraryHandle h, double bpm);
Set the global tempo

---

double GP_GetBPM(LibraryHandle h);
Get the global tempo

---

void GP_Previous(LibraryHandle h);
Move to previous variation or rackspace

---

void GP_Next(LibraryHandle h);
Move to next variation or rackspace

int GP_GetChordProFilenameForSong(LibraryHandle h, int atSongIndex, char* returnBuffer, int bufferLength);
Get the full path to a ChordPro file associated with the given song index of the current setlist

---

int GP_GetSetlistCount(LibraryHandle h);
Return the number of setlists in the gig file

---

int GP_GetSetlistName(LibraryHandle h, int atSetlistIndex, char* returnBuffer, int bufferLength);
Get the name of the setlist at the given index

---

int GP_GetCurrentSetlistIndex(LibraryHandle h);
Get the index of the current setlist

---

bool GP_SwitchToSetlist(LibraryHandle h,int setlistIndex);
Switch to the setlist defined by the index

---

int GP_GetRackspaceUuid(LibraryHandle h, int atIndex,
char* returnBuffer, int bufferLength);
Get the uuid for the rackspace atIndex. Note - use an index of -1 to get the uuid for the global rackspace

---

int GP_GetSongUuid(LibraryHandle h, int atIndex,
char* returnBuffer, int bufferLength);
Get the uuid for the song atIndex.

---

bool GP_MapWidgetToPluginParameter)(LibraryHandle h, const char* widgetName, const char* pluginHandle, int parameterNumber, bool useGlobalRackspace);
Map a widget to a specific plugin and parameter number. NB this is an asynchronous operation as it must be executed on the GP GUI thread.

Callbacks list

Following is a list of callbacks that GP can invoke to which your library can respond. You must register the callback by name to be able to receive these callbacks.

```
bool OnMidiIn(const char*  deviceName, const uint8_t* data, int length);
```

Returns whether the midi event should be passed on to the next handler

---

```
void OnClose();
```

Called when Gig Performer is about to shut down

```
void OnEditStateChanged(bool inEditState);
```

Called when user enters or exits widget editing state.

```
void OnSwitchToWiringView();
```

called when the user switches into the plugin wiring view

```
void OnSwitchToPanelView();
```

Called when the user switches into the panel view mode. Note that if you are in the wiring view and you switch directly to setlist mode, there will also be a switch back to panel view call

---

```
void OnModeChanged(int mode);
```

Called when the user switches between Panel/Wiring (0) and Setlist (1) mode

---

```
void OnStatusChanged(GPStatusType status);
```

Called when something changes in GP for which the external API can be notified. The current items are defined in GPTypes.h of the C interface

---

```
void OnMidiDeviceListChanged(const char** inputs, int intputCount,
                             const char** outputs, int outputCount);
```

Called when a MIDI device is connected or disconnected from the computer. You get an array of strings for the input names and for the output names

---

```
void OnOpen();
```

Called when Gig Performer has finished initializing itself

```
void OnRackspaceActivated();
```

Called when user switches to a new rackspace

```
void OnSongChanged(int oldIndex, int newIndex);
```

Called when user switches to a new song in setlist mode

```
void OnSongPartChanged(int oldIndex, int newIndex);
```

Called when we switch to a new song part

void OnTunerModeChanged(bool visible);
Called when the user switches in or out of tuner mode

---

void OnGlobalPlayStateChanged(double playing);
Called when the global play mode switches: 0.0 => off, 1.0 => on, 0.5 => on via Ableton Link

---

void OnVariationChanged(int oldIndex, int newIndex);
Called when user switches from one variation to another

---

void OnWidgetValueChanged(const char* widgetName, double newValue);
Called when a widget is turned or otherwise adjusted. The library must register its interest in listening for such changes.

---

void OnWidgetCaptionChanged(const char* widgetName, const char* newValue);
Called when a widget caption is changed. The library must register its interest in listening for such changes.

---

void OnWidgetStateChanged(const char* widgetName, int newState);
Called when a widget handle is defined (newState = 0) or removed (newState = 1). The library must register its interest in listening for such changes.

---

void OnSetlistChanged(const char* newSetlistName);
Called when the user switches the setlist. The library must register its interest in listening for such changes.

---

void OnTempoChanged(double newBPM);
Called when the tempo changes (duh!). The library must register its interest in listening for such changes.

Optional functions that your library can provide

Panel support

If your library includes resources such as predefined front panels, you will need to implement the following three functions so that Gig Performer can query your library to get access to the panels that are available through your library. If you are working in C++, make sure you enclose these functions with **extern "C"** so as to ensure the appropriate calling conventions are used.

EXPORTED int GetPanelCount();
Return the number of panels available from this library

EXPORTED int GetPanelName(int index, char* buffer, int bufferLength);
Fill the buffer with the name of a panel at the specified index up to the maximum buffer size. The function returns the number of bytes that would be needed for the full panel name and you can call the function again with a larger buffer if necessary.

EXPORTED int GetPanelXML(int index, char* buffer, int bufferLength);
Fills the buffer with a chunk of XML code representing a predefined panel. You can create these by saving a Gig Performer panel. The function returns the number of bytes that would be needed for the full panel name and you can call the function again with a larger buffer if necessary.

Menu support

Gig Performer allows your library to register menu items that a user can select to invoke functionality implemented in your library. You will need to implement the following three functions for this feature to work. If you are working in C++, make sure you enclose these functions with **extern "C"** so as to ensure the appropriate calling conventions are used.

EXPORTED int GetMenuCount();
Return the number of menu items available from this library

EXPORTED int GetMenuName(int index, char* buffer, int bufferLength);
Fill the buffer with the name of a menu at the specified index up to the maximum buffer size. The function returns the number of bytes that would be needed for the full menu name and you can call the function again with a larger buffer if necessary.

EXPORTED void InvokeMenu(int index);
Invokes the code in your library associated with the provided menu index.

EXPORTED int RequestGPScriptFunctionSignatureList(GPScript_AllowedLocations location, ExternalAPI_GPScriptFunctionDefinition* *list);
Inject GP Script functions into Gig Performer to allow users to execute code implemented by the extension developer. See below for more details on this topic

The C++ Wrapper

While the underlying C-style interface will allow developers to use almost any language they wish provided there is a C interface to it, many developers are familiar with C++ and so we provide some C++ classes around the underlying C interface to make it easier to get started. The C++ wrappers include some useful support classes to faciliate the creation of MIDI messages, particularly sysex messages from strings as well as some other generally useful functions. See the contents of the folder GP_SDK/interfaces/CPP for more information.
To use the C++ wrapper, the only file you need to modify is called LibMain.h which must inherit from GigPerformerAPI (defined below). You need only implement the methods for events in which you are interested.

GigPerformerAPI

This class defines all the callbacks to which you can respond.

```
class GigPerformerAPI : public GigPerformerFunctions
{
   public:
      // This MUST be defined in your class to describe your product
      virtual std::string GetProductDescription() = 0;
      // Use this to register callbacks and to do any other setup you need
      virtual void Initialization();
      virtual void OnEditStateChanged(bool inEditState);
      virtual void OnModeChanged(int mode);
      virtual void OnSwitchToPanelView();
      virtual void OnSwitchToWiringView();
      virtual void OnTunerModeChanged(bool visible);
      virtual void OnGlobalPlayStateChanged(bool playing);
      virtual void OnWidgetValueChanged(const std::string & widgetName,
                                        double newValue);
      virtual void OnWidgetCaptionChanged(const std::string & widgetName,
                                        const std::string & newValue);
      virtual void OnWidgetStateChanged(const std::string & widgetName,
                                        int newState);
      virtual void OnMidiDeviceListChanged(std::vector< std::string> & inputs,
                                           std::vector< std::string> & outputs);
      virtual bool OnMidiIn(const std::string & deviceName,
                            const uint8_t* data, int length);
      virtual void OnStatusChanged(GPStatusType status);
      virtual void OnSetlistChanged(const std::string & newSetlistName);

      virtual void OnSongChanged(int oldIndex, int newIndex);
      virtual void OnSongPartChanged(int oldIndex, int newIndex);
      virtual void OnRackspaceActivated();
      virtual void OnVariationChanged(int oldIndex, int newIndex);
      virtual void OnOpen();
      virtual void OnClose();
```

```
   public:
      virtual int GetPanelCount();
      virtual std::string GetPanelName(int index);
      virtual std::string GetPanelXML(int index);

      virtual int GetMenuCount();
      virtual std::string GetMenuName(int index);
      virtual void InvokeMenu(int itemIndex);
   public:
      virtual int RequestGPScriptFunctionSignatureList(GPScript_AllowedLocations
location, ExternaAPI_GPScriptFunctionDefinition* *list);

   public:
      GigPerformerAPI(LibraryHandle handle) : GigPerformerFunctions(handle) {}
      virtual ~GigPerformerAPI() {}
};
```

This class inherits from a parent class, GigPerformerFunctions, which defines all the functions your library can call to access/manipulate Gig Performer itself.

GigPerformerFunctions

```
class GigPerformerFunctions
{
public:
   GigPerformerFunctions(LibraryHandle handle);
   virtual ~GigPerformerFunctions();

   bool registerCallback(const std::string & callbackName);
   bool unregisterCallback(const std::string & callbackName);

   void setPlayheadState(bool play);
   bool getPlayheadState();

   void showTuner(bool show);
   bool tunerShowing();

   void enableMetronome(bool enable);
   bool metronomeEnabled();

   void switchToSetlistView();
   void switchToWiringView();
   void switchToPanelView();

   void getPluginList(std::vector<std::string> & list , bool useGlobalRackspace);
```

```
   bool pluginExists(const std::string & pluginHandle, bool useGlobalRackspace);

   void setPluginParameter(const std::string & pluginHandle, int parameterNumber,
double value , bool useGlobalRackspace);
   double getPluginParameter(const std::string & pluginHandle, int parameterNumber,
bool useGlobalRackspace);
   int getPluginParameterCount(const std::string & pluginHandle, bool
useGlobalRackspace);
   std::string getPluginParameterName(const std::string & pluginHandle, int
parameterIndex, bool useGlobalRackspace);
   std::string getPluginParameterText(const std::string & pluginHandle, int
parameterIndex, bool useGlobalRackspace);

   void getWidgetList(std::vector<std::string> & list, bool useGlobalRackspace );
   bool widgetExists(const std::string & widgetName);
   double getWidgetValue(const std::string & widgetName);
   bool setWidgetValue(const std::string & widgetName, double newValue);
   bool  resetWidgetToDefault(const std::string & widgetName, double newDefault = -1);
   bool setWidgetCaption(const std::string &  widgetName, const std::string &
newCaption);
   std::string getWidgetCaption(const std::string &  widgetName);
   int   RGBAToColor(double red, double green, double blue, double alpha);
   void  ColorToRGBA (int color, double & red, double & green, double & blue, double &
alpha);

   void setWidgetFillColor( const std::string & widgetName, int color);
   void setWidgetOutlineColor(const std::string & widgetName, int color);
   void setWidgetOutlineThickness(const std::string & widgetName, int thickness);
   void setWidgetOutlineRoundness(const std::string & widgetName, int roundness);
   int  getWidgetFillColor( const std::string & widgetName);
   int  getWidgetOutlineColor(const std::string & widgetName);
   int  getWidgetOutlineThickness(const std::string & widgetName);
   int  getWidgetOutlineRoundness(const std::string & widgetName);
   int  setWidgetBounds(const std::string & widgetName, int left, int top, int width,
int height);
   int  getWidgetBounds(const std::string & widgetName, int & left, int & top, int &
width, int & height);

   void setWidgetHideOnPresentation(const std::string &  widgetName, bool hide);
   bool getWidgetHideState(const std::string &  widgetName);



   std::string getRackspaceUuid(int atIndex);
   std::string getSongUuid(int atIndex);
```

```
   void mapWidgetToPluginParameter(const std::string & widgetName, const std::string
& pluginHandle, int parameterNumber, bool useGlobalRackspace);

   void  panic();
   bool  listenForWidget(const std::string & widgetName, bool listen);
   bool  listeningForWidget(const std::string & widgetName);

   bool  listenForMidi(const std::string & deviceName, bool listen);
   bool  listeningForMidi(const std::string & deviceName);
   int  getMidiInDeviceCount();
   std::string  getMidiInDeviceName(int index);
   int  getMidiOutDeviceCount();
   std::string  getMidiOutDeviceName(int index);
   void  sendMidiMessageToMidiOutDevice(const std::string & deviceName, const uint8_t*
midiData, int length);
   void  sendMidiMessageToMidiOutDevice(const std::string & deviceName, std::string &
midiData);
   void sendMidiMessageToMidiOutDevice(const std::string & deviceName, const
GPMidiMessage & message);
   std::string  textToHexString(const std::string & text);
   void  injectMidiMessageToMidiInputDevice(const std::string & deviceName, const
uint8_t* midiData, int length);
   void  injectMidiMessageToMidiInputDevice(const std::string & deviceName,
std::string & midiData);
   void  injectMidiMessageToMidiInputAlias(const std::string & rigManagerAlias, const
uint8_t* midiData, int length);
   void  injectMidiMessageToMidiInputAlias(const std::string & rigManagerAlias,
std::string & midiData);

   int  getSongCount();
   std::string GetSongName(int atIndex);
   std::string GetArtistName(int atIndex);
   int  getCurrentSongIndex();
   std::string getVariationNameForSongPart(int atSongIndex, int atPartIndex);
   int  getSongpartCount(int atSongIndex);
   std::string getSongpartName(int atSongIndex, int atIndex);
   int  getCurrentSongpartIndex();
   bool inSetlistMode();
   bool switchToSong(int songIndex, int partIndex);
   bool switchToSongPart(int partIndex);

   void consoleLog(const char* message);
   void consoleLog(const std::string & message);
   void scriptLog(const char* message, bool openLogWindow);
   void scriptLog(const std::string & message, bool openLogWindow);
   std::string getInstanceName(int atIndex);
   void getCurrentTimeSignature(int & numerator, int & denominator);
```

```
   int getRackspaceCount();
   std::string getRackspaceName(int atIndex);
   int getCurrentRackspaceIndex();
   int getCurrentVariationIndex();
   int getVariationCount(int atRackspaceIndex);
   std::string getVariationName(int atRackspaceIndex, int atIndex);
   bool switchToRackspace(int rackspaceIndex, int variationIndex = 0);
   bool switchToRackspaceName(const std::string & rackspace, const std::string &
variation = "");
   bool switchToVariation (int variationIndex = 0);


   bool saveGigUnconditionally(bool withTimestamp);
   bool loadGigByIndex(int indexNumber);

   void setBPM(double bpm);
   double getBPM();

   void tap();
   void previous();
   void next();
   std::string getChordProFilenameForSong(int atIndex);
   int getSetlistCount();
   std::string getSetlistName(int setlistIndex);
   int getCurrentSetlistIndex();
   bool switchToSetlist(int setlistIndex);


private:
   LibraryHandle fHandle;
};
```

Using GP Script functions with extensions - experimental

The GP API includes a mechanism that allows extension developers to define GP Script functions that are available to Gig Performer users. The implementations of those functions exist within the extension itself. The purpose of this mechanism is to allow third parties to develop extensions such as graphic packages, algorithmic music packages, lighting control systems and so forth that can then be manipulated inside a GP Script callback while performing with Gig Performer.
This process must be tested extremely carefully as erroneous implementations can easily crash Gig Performer. The extension developer must insure that incoming parameters are within expected ranges so that a Gig Performer user cannot crash the system by sending invalid values in a GP Script function call.

Types

The required types you will need to support GP Script are defined in GPTypes.h which is included in the SDK

```
typedef
   void*
      GPRuntimeEngine; // Opague type needed for GPScript helper functions
```

The helper functions used to accept and send back parameters will use instances of this type.

```
typedef
   void (*TGPScriptExecutableFunctionSignature)(GPRuntimeEngine* vm);
```

This is the required signature for the functions you implement in the extension that can be called from Gig Performer via GP Script

```
typedef
   struct
      {
         const char* functionName; // Just the name of the function
         const char* args; // The args and the optional return type
         const char* returns; // optional return type clause
         const char* description; // For the helper in the script editor
         TGPScriptExecutableFunctionSignature functionToExecute;
      }
      ExternaAPI_GPScriptFunctionDefinition;
```

This structure is used to define the parts of a GP function so that it can be properly injected back into Gig Performer. During initialization, the function `RequestGPScriptFunctionSignatureList` will be called and you must provide an array of this structure, appropriately filled in. An example of this is included in the **cpp** example in the SDK. Gig Performer also includes a location value that indicates what script entity is asking for your function list. This allows you to control whether your functions should be available everywhere or perhaps only in the Gig Script, or only in the Global Rackspace, etc.

Accessing parameters and returning results

Arguments defined in the function are pushed on to a stack and the developer must pull all the arguments from the stack as the initial action in any function defined in the extension that's associated with an injected GP Script function signature. In particular, the developer must pull the arguments from the stack in the reverse order.

For example, suppose you have defined the following GP Script signature to multiply the first two arguments, add in the third argument and return the result:

```
{"MA", "a:double, b:double, c:double", "Returns Double", "Mult And Add", MultAdd}
```

This corresponds to the GP Script function that the GP user will see

```
function MA(a : double, b : double, c : double) Returns Double
```

The function would most likely be implemented in your extension as follows:

```
extern "C" void MultAdd(GPRuntimeEngine* vm)
{
  // Access the incoming arguments in the reverse order
  double c = GP_VM_PopDouble(vm);
  double b = GP_VM_PopDouble(vm);
  double a = GP_VM_PopDouble(vm);

  // Perform your calculation
  double result = a * b + c;

  // Push the result back on to the stack
  GP_VM_PushDouble(result);
}
```

If you do not perform the three "pops" at the beginning (in the reverse order of the declared function) and the "push" at the end, the GP Script stack will be corrupted and Gig Performer will crash.

GP Script supported function types

Currently, the only types that are available are the basic scalar types, i.e, Integer, Double, Boolean and String. A future version will provide support for arrays and other objects.