Interfacing With Smart Graphics Media Player

Introduction:

Crestron introduced a new easy to program Media Player Smart Object. This object eliminated the need for multiple pages and 100+ signals in VTPro and SIMPL Windows for each audio device. This was achieved by using one serial signal carrying all information needed to configure the UI. This API will demonstrate how to interface with the Media Player Smart Object using that serial signal.

Crestron RPC:

Crestron RPC (CRPC) is a service that allows remote objects to communicate with each other. CRPCService is the main class to be used for CRPC. Available is a wrapper called CRPCConnection over the CRPCService which provides proper networking support on Crestron control systems. The CRPCConnection is the main class for developers to use. It expands upon several methods from the CRPCService to support multi-homed control systems.

At times, the developer may need to reference the CRPCService class for some of its services. A particular service supported by the CRPCService is one with post processing capabilities. In order to provide correctly sequenced messages to be sent out to remote objects, the CRPCService provides a method to add a post processing delegate. After a method returns from the local CRPC object and the CRPCService sends out the results, the CRPCService will then execute the delegate.

Steps to Interface:

* Note: The examples below are code snippets from the MediaPlayerSDK solution. Please refer to it for further implementation details.

1. Create a SIMPL# solution and class.

Solution:	MediaPlayerSDK
Class:	MP_SDK.cs

Implement the Media Player API: (see below for full API)

1. Create a Media Player object inheriting from IMediaPlayer. See “Identifying with CRPC” for more details.

   ```
   public class DeviceMediaPlayer : IMediaPlayer
   ```

   a. Set this new IMediaPlayer object to identify as a CRPCObject. See “Identifying with CRPC” for more details.

      ```
      [Crestron.CRPC.Common.CrpcObject ("E4838A12-1198-49e6-BE07-0F3CE8BA4911 ")]
      public class DeviceMediaPlayer : IMediaPlayer
      ```

2. Create a Menu object inheriting from IMenu.

   ```
   public class DeviceMediaMenu : IMenu
   ```

   a. Set this new IMenu object to identify as a CRPCObject. See "Identifying with CRPC" for more details.

```
[Crestron.CRPC.Common.CrpcObject("1ABE04BE-2BD0-4d29-BCC5-672DC80F79B3")]
public class DeviceMediaMenu : IMenu
```

Set Up Communications Between SIMPL+ and SIMPL#

3. Define delegate types for sending data to SIMPL+

```
public delegate void DelegateFnString(SimplSharpString myString);
```

Within Your Media Player object:

4. Instantiate a Crestron RPC service within the Initialize class.

```
private CRPCConnection Connection = new CRPCConnection();
```

5. Create an Initialize function taking in three parameters. This function will be used to set up call back functions and the CRPC service.

```
public void Initialize(ushort port, ushort adapterID, int connectionType) {}
```

   ushort port
   - the port number to use for connection

   ushort adapterID
   - identifies on which subnet of the control system the device is on (0 – LAN, 1 – Control Subnet)

   int connectionType
   - identifies whether to use join or direct connection (0 – Join, 1 –Direct)

6. Define a SIMPL# callback function for the Crestron RPC.

```
public void myTransport_Sendback(string stream)
{
        if (MessageOut != null)
         {
                MessageOut(new SimplSharpString(stream));
        }
}
```

   Where MessageOut is the delegate defined:

```
public DelegateFnString MessageOut { get; set; }
```

7. Define a SIMPL# function for SIMPL+ to send in data within the Initialize class.

```
 public void MessageIn(string pkt)
{
        CRPCService.OnJoinData(pkt);
}
```

8. Now assign the SIMPL# callback function to a variable for identification with the Crestron RPC Service.

```
var sendback = new simplsharpstringcallback(myTransport_Sendback);
```

9. Initialize the Crestron RPC service

```
if (connectionType == 0)    // use only the join connection
            CRPCService.Initialize(sendback, false);
else if (connectionType == 1)   // use direct connection as default
            CRPCService.Initialize(sendback, port, adapterID, false, false);
```

10. Instantiate a Menu object. For this implementation, there will always be n+1 menus available where n is the number of clients (Media Player Smart Objects) registered with the Crestron RPC Service. Subsequent menus will be created later.

```
DeviceMediaMenu MenuFirst = new DeviceMediaMenu();
```

11. Store the newly created menu name for later use

```
UnusedMenu = MenuFirst.Name;
```

12. Next, the Media Player and Menu objects need to be registered with the Crestron RPC Service

```
CRPCService.AddObject<DeviceMediaPlayer>(this, this.Name, true);
CRPCService.AddObject<DeviceMediaMenu>(MenuFirst, MenuFirst.Name, true);
```

13. You have now built the minimum for sending and receiving between SIMPL+, SIMPL#, and the Crestron RPC library.

Within Your Menu object:

To provide unique instances of the Menu object to each client, n + 1 menus are created. A menu instance is associated with a client when the client requests a menu. In the Media Player API, the client requests a menu using the GetMenu method. Therefore, the following is used to assign and create a new menu.

```
[CrpcMember("GetMenu")]
public Dictionary<string, string> GetMenu(string uuid)
{
            Dictionary<string, string> instanceName = new Dictionary<string, string>();
            try
            {
                        ErrorLog.Notice("getting menu!!! new menu: {0} and {1}",
                        uuid, ClientMenu.ContainsKey(uuid));
            }
            catch
            {
                        ErrorLog.Notice("ERROR");
```

```
		}

		if (ClientMenu.ContainsKey(uuid))
		{
			ErrorLog.Notice("Found key: {0}", uuid);
			instanceName.Add("instanceName", ClientMenu[uuid]);
			ErrorLog.Notice("instanceName: {0}", ClientMenu[uuid]);
		}
		 else
		{
			DeviceMediaMenu Menu = new DeviceMediaMenu();
			instanceName.Add("instanceName", UnusedMenu);
			ClientMenu.Add(uuid, UnusedMenu);
			mediaMenuList.Add(Menu);
			UnusedMenu = Menu.Name;
			CRPCService.AddChildObject<DeviceMediaMenu>(Menu, Menu.Name,
			this.Name, true);
			Menu.InitializeMenu(this);
		}
		return instanceName;
	}
```

Identifying with Crestron RPC:

Crestron RPC Service allows objects to be registered (ex. server object) and served up to clients. To accurately identify the implementation of each method, property, and event with Crestron RPC Service, each must be tagged with the following attributes:

- For Objects:

  ```
  [Crestron.CRPC.Common.CrpcObject("GUID")]
  ```

  Where GUID is a uniquely generated UUID by the programmer

- For Methods:

  ```
  [CrpcMember("method")]
  ```

  Where method is the name of the method defined in Creston’s IMediaPlayer/IMenu interfaces

- For Properties:

  ```
  [CrpcMember("property")]
  ```

  Where property is the name of the property defined in Creston’s IMediaPlayer/IMenu interfaces

- For Events:

  ```
  [CrpcMember("event")]
  ```

  Where event is the name of the method defined in Creston’s IMediaPlayer/IMenu interfaces

Without these tags, the methods, properties, and events will not be available to clients.

Upgrading from SDK 1.2:

If you have a previous project, it can be used without changes. However, if you wish to accurately represent the changes after 1.2, the following needs to be updated.

The classes were copied from Crestron.CRPC.Common to Crestron.CRPC.MediaPlayer. The original classes will remain in Crestron.CRPC.Common for backwards compatibility for the time being.

1. Replace `using Crestron.CRPC.Common;` with `using Crestron.CRPC.MediaPlayer;`
   Only one of these statements may be used to avoid namespace conflicts.

2. Replace `[CrpcObject("E4838A12-1198-49e6-BE07-0F3CE8BA4911")]`
   with `[Crestron.CRPC.Common.CrpcObject("E4838A12-1198-49e6-BE07-0F3CE8BA4911")]`
   CrpcObject had to remain within the Common namespace. Other attributes can be left as is.

3. Replace `CrpcEventHandler` with `MediaPlayerCrpcEventHandler`

4. Update your classes to represent the changes within the interfaces. See the SDK document for a list of currently supported methods, properties, and events.

SDK Examples

Status Messages:

Status Messages are used to provide a popup to the client. The Status Messages contain a message and may also contain a text input box or a keypad and up to 5 buttons. The text input may have a default value, and the message can have a timeout. When the user has selected an input, dismissed the popup, or the popup has timed out, the client will return a response message.

1. Creating a Status Message

```
MenuStatusMsg msgItems = new MenuStatusMsg();
msgItems.text = "What would you like to do";
msgItems.timeoutSec = 30;
msgItems.userInputRequired = "confirmation";
msgItems.initialUserInput = "";
msgItems.show = true;
msgItems.textForItems = new string[3] { "Save", "Clear", "Cancel" };
msgItems.localExit = false;

StatusMsgMenu = msgItems;
TriggerStatusMsgMenuChangedEvent(msgItems);
```

2. StatusMsg vs StatusMsgMenu
   Both the now playing and menu objects can dispatch a Status Message. The Now Playing object is shared across all clients, so all user interfaces connected to the source will display the popup. The Menu objects can be configured to be client specific, allowing a single UI to be targeted with a Status Message.

   Now Playing will use StatusMsg for the property, StatusMsgChanged for the event, and StatusMsgResponse for the callback from the client.

   Menu will use StatusMsgMenu for the property, StatusMsgMenuChanged for the event, and StatusMsgResponseMenu for the callback from the client.

3. Property vs Event
   The StatusMsgChanged and StatusMsgMenuChanged events alert connected clients of a new Status Message. If the events are received with the show field set to true, the client will display its modal to the user.

   The StatusMsg and StatusMsgMenu properties allow a client to retrieve the popup when it connects or when the source is selected. By leaving the show property true in the property, an ongoing message can be sent to a client. Clients will continue to show the popup on each connection or source selection until the property's show field is set false.

   It is not necessary to set the property for single-use popups. The event can be triggered with the appropriate Status Message data independent of setting the property.

List Specific Functions

List specific functions allow the Menu UI to show buttons that trigger specific actions. The ListSpecificFunctions property can be updated at any time, and the StateChanged event will alert the UI of a change in the availability of each function. The following are currently supported:

Function	Appearance	Description
PlayAll	⊡	Plays all items in the current menu
Create	+	Used to start the station creation process
Find	🔍	Used to find an item in the current menu. Method receives a query string and returns the index of the closest matching item.
QuickList	♪	Used to quickly display a list of presets or the current queue
Advanced	🎚	Used to display configuration options
BackToTop	🏠	Returns the menu to the highest level in the current hierarchy
Favorites	❤	Used to display a list of favorite songs/stations

To update the list specific functions:

```
List<string> functions = new List<string>();

if (MP.Data.myMenu[menuStack.Peek()].Searchable == "True")
{
    functions.Add("Find");
    FindAvailable = true;
}
else
    FindAvailable = false;

if (MP.Data.myMenu[menuStack.Peek()].Editable == "True")
{
    functions.Add("Create");
}

if (MP.Data.myMenu[menuStack.Peek()].PlayAll == "True")
{
    functions.Add("PlayAll");
}

SetListSpecificFunctions(functions.ToArray());
TriggerStateChangedEvent();
```