Control 4 Media Service Proxy

Driver Development Documentation
OS 3.2.1

License, Copyright, and Trademark

The content contained in this repository is the intellectual property of Snap One, LLC, (formerly known as Wirepath Home Systems, LLC), and use without a valid license from Snap One is strictly prohibited. The user of this repository shall keep all content contained herein confidential and shall protect this content in whole or in part from disclosure to any and all third parties except as specifically authorized in writing by Snap One.

License and Intellectual Property Disclaimer

The content in this repository is provided in connection with Snap One products. No license, express or implied, by estoppal or otherwise, to any intellectual property rights is granted by this document or in this repository. Except as provided in Snap One's terms and conditions for the license of such products, Snap One and its affiliates assume no liability whatsoever and disclaim any express or implied warranty, relating to the sale and/or use of Snap One products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Snap One products are not intended for use in medical, lifesaving, or life sustaining applications.

Information regarding third-party products is provided solely for educational purposes. Snap One is not responsible for the performance or support of third-party products and does not make any representations or warranties whatsoever regarding the quality, reliability, functionality or compatibility of these products. The reader is advised that third parties can have intellectual property rights that can be relevant to this repository and the technologies discussed herein, and is advised to seek the advice of competent legal counsel regarding the intellectual property rights of third parties, without obligation of Snap One.

Snap One retains the right to make changes to this repository or related product specifications and descriptions in this repository, at any time, without notice. Snap One makes no warranty for the use of this repository and assumes no responsibility for any errors that can appear in the repository nor does it make a commitment to update the content contained herein.

Copyright

Trademarks

Snap One and Snap One Logo, Control4 and the Control4 logo, and DriverWorks are trademarks or registered trademarks of Snap One, LLC. Other product and company names mentioned in this repository may be the trademarks or registered trademarks of their respective owners.

Derivative Works

To the extent that you create any "Derivative Work" (meaning any work that is based upon one or more preexisting versions of the work provided to you in this repository, such as an enhancement or modification, revision, translation, abridgement, condensation, expansion, collection, compilation or any other form in which such preexisting works may be recast, modified, transformed or adapted, explicitly including without limitation, any updates or changes to Snap One, LLC's software code or intellectual property) such Derivative Work shall be owned by Snap One, LLC and all right, title and interest in and to each such Derivative Work shall automatically vest in Snap One, LLC. To the extent any Derivative Work does not automatically vest in Snap One, LLC by operation of law, you hereby assign such Derivative Work to Snap One, LLC with full title guarantee. Snap One, LLC shall have no obligation to grant you any right in any such Derivative Work.

Contact Us

Snap One, LLC
11734 S. Election Road
Salt Lake City, UT 84020 USA
http://www.control4.com

Updates and Modifications to this Version of the Document

What's New in OS 3.2.1
A new Capability called list_nowplaying_non_c4 has been added. This capability indicates an MSP driver that does not use Control4's DIGITAL_AUDIO.

A new Capability called list_no_nowplaying has been added. This capability indicates an MSP driver that should never automatically jump to the Now Playing screen when a favorite is selected.

The list_no_auto_select Capability released in 3.2.0 has been deprecated.

What's New in OS 3.2.0
A new Capability called list_no_auto_select has been added. This capability is included primarily to support the Neeo remote. When browsing into an MSP list that has only a single 'leaf node' item ('leaf node' means the item would make a selection, as opposed to browsing into another list), the Neeo will automatically select the item, to eliminate a button press.

What's New in OS 3.1.2
There were no modifications to the document in conjunction with the OS 3.1.2 release.

What was New in OS 3.1.0
DeatailType Screen
Information regarding the DetailType Screen has been added to this version of the documentation. The screen provides the ability to present further details regarding media when the user selects a "View Details" menu option.

What was New in OS 3
Favoriting
An end user's ability to designate Navigator UI screens as "Favorites" is a feature introduced in Operating System OS 3. In conjunction with this, drivers written using the Media Service Proxy can provide the ability for end users to designate favorite media delivered through an MSP driver such as Playlists, Albums, Tracks, Stations, Movies, TV Shows, etc. More information can be found in the Appendix section of this document.

Driver Notification Context Example
New code examples on have been added to the Media Service Proxy Driver Notification Prototype section in the Appendix.

DashboardChanged Event
The DashboardChanged event now includes content on the use of custom icons and actions.

What was New in 2.10.X
CUSTOM_SELECT
An XML param type called CUSTOM_SELECT has been added. Its use provided the ability of a Media Service Proxy driver to deliver a browse-able list of media related elements.

MSP by Chapter
Beginning with Operating System 2.10.0, a new documentation set has been included to assist with driver development using the Media Service Proxy. The MSP by Chapter content is designed to instruct on driver development in an incremental manner. Chapter 1 includes

documentation and a sample driver explaining the Now Playing functionality supported through the Proxy.

This content can be accessed from within the SDK at the following location:
DriverWorks SDK\Media Service Proxy Resources\Documentation\MSP by Chapter

Additional Chapters will be released in the future.

This page is intentionally blank.

Overview of the Media Service Proxy

Key Features and Benefits

Control4's Media Service Proxy (MSP) provides a layer of commands, notifications, events and other data handling elements that will support the development of drivers for media-based services and devices. This iteration of a media-focused proxy represents a new direction in driver development as it uses Control4's .c4z driver architecture and offers the ability to support XML-based user interfaces.

From an end user perspective, drivers written against the Media Service Proxy will be able to provide a user interface that can browse cloud based media services consistently across numerous navigator devices. The proxy is designed to be platform independent with regards to devices displaying its user interface while maintaining the ability to easily integrate with new clouds or devices introduced to the Control4 system.

Pre-requisites of Use

Control4 OS
The Media Service Proxy was designed to be compatible with Operating System 2.6.0 and later. This OS requirement is applicable to ComposerPro and all devices using the Media Service Proxy.

c4z. Driver Architecture
The Media Service Proxy was designed to be used in conjunction with Control4's .c4z driver architecture.

Media Service Proxy Schema Document
Throughout this document, we will refer to the Media Service Schema document (MediaServiceProxyUI.xsd) to further explain a concept or cite a section of the schema document for code review. Having this document readily available is recommended when working with the Media Service Proxy. The schema document can be opened in any text editor.

Hello World Sample Driver
Throughout this document, we will refer to the sample Hello World driver included with the Media Service Proxy to further explain a concept or cite section of the sample driver's code for review. Having this driver readily available is recommended when working with the Media Service Proxy.

Driver Developer Requirements
The Media Service Proxy is one of Control4's more complex Proxies. Creating device drivers which will rely on the Proxy will require knowledge of the Control4 system architecture, Control4 Driver Development best practices, Lua etc.

For driver development information and training materials, please see the Control4 University at: http://go.bluevolt.com/control4/Home/
Once logged in, search on "Driver Development."

How to Use this Document

This document is organized in a manner that provides overview information followed by detailed examples on how to build a driver to handle data and render UI solutions through the Media Service Proxy. It is recommended that the document be read prior to beginning driver development efforts.

The Media Service Proxy represents a departure from previously delivered Control4 proxies. It not only fulfills the role of a traditional proxy, but it also supports the ability to render custom user interfaces as it delivers data to and from Navigator devices. The content in the overview sections of this document detail how the proxy accomplished this within the Control4 driver architecture. Understanding these concepts is significant as the development areas of the document assume this level of knowledge.

The Understanding Screens section moves from driver development theory to application. This section outlines the steps required to design navigator screens based on the Media Service Proxy to appropriately display data from media devices and services. It will also explain how these screens handle user interaction to fetch data or move through the navigator based driver experience. These development exercises will reference a sample driver included with the Media Service Proxy named Hello World.

After you've learned how to design screens to display the data, we'll look at how the data itself is handled by the screens and discuss each of the data flow layers the data passed through. Again, the Hello World sample driver will be used for these explanations.

The next section of the document discusses the use of two utilities included with the Media Service Proxy: Schema Validator and CreatePO. Schema Validator is useful in validating the XML created to support a driver against the defined Media Service Proxy Schema: MediaServiceProxyUI.xsd

The CreatePO utility can assist with the creation of .po files for use in localizing the UI elements containing text that comes through the Media Service Proxy.

Finally, the end of the document includes an Appendix section which serves as a reference guide for driver development elements such as proxy commands, notifications, capabilities and events. In addition to this content, an overview of Control4's new driver format is included in: *An Overview of the .c4z File Format.* Contol4 expects all driver using the Media Service proxy to be delivered in a .c4z file.

Finally, a Control4 glossary is also included.

Note that throughout this document Media Service proxy may be referred to with the acronym: MSP.

The Hello World.c4z Driver

The purpose of Hello World.c4z is to serve as a tutorial driver in support ofControl4's Media Service Proxy. It is intended to be used in conjunction with the Media Service Proxy Driver Development documentation. The driver and its included functionality are referenced extensively through the documentation and it is recommended that that driver be loaded into an environment and used while the documentation is referenced.

Hello World.c4z offers a "user experience" which highlights the features available to driver developers who wish to create drivers which leverage Control4's Media Service Proxy. Among the elements highlighted in the driver are Tabs, Filters, Icons, Media Artwork as well examples of the screens supported through the Media Service Proxy.

Hello World represents a driver written to support a media service. Examples will be forthcoming to represent a driver supporting a media device. The driver contains several hard coded radio channels which offer an example on how to create a driver which connects to radio streaming service. The audio played through the driver when radio stations are selected is self-contained within the Hello World code and is provided for example purposes only.

The driver also contains several albums and songs. The code used to play this media provides an example of streaming audio from a device or service. The album audio played through the Hello World driver is self-contained within the driver and is for demonstration purposes only. The songs clips provided are 10 second in length to minimize the size of driver.

The Hello World driver offers a "Settings" screen which contains numerous configuration elements that are offered through the Media Service Proxy. These elements can also be seen and configured in the driver's properties list in ComposerPro.

The Code samples referenced in the Media Service Proxy Driver Development documentation are derived directly from the driver.lua and driver.xml portions of the Hello World driver. All attempts were made to include code samples which can be copied from the document and exercised in the Hello Word's Lua window or pasted into a text editor.

Information found within this document is subject to change. This includes code samples, screen images and supporting textual content. The driver and supporting documentation is intended for internal review and partner feedback only.

The HW Selected Device and Streaming Device Drivers

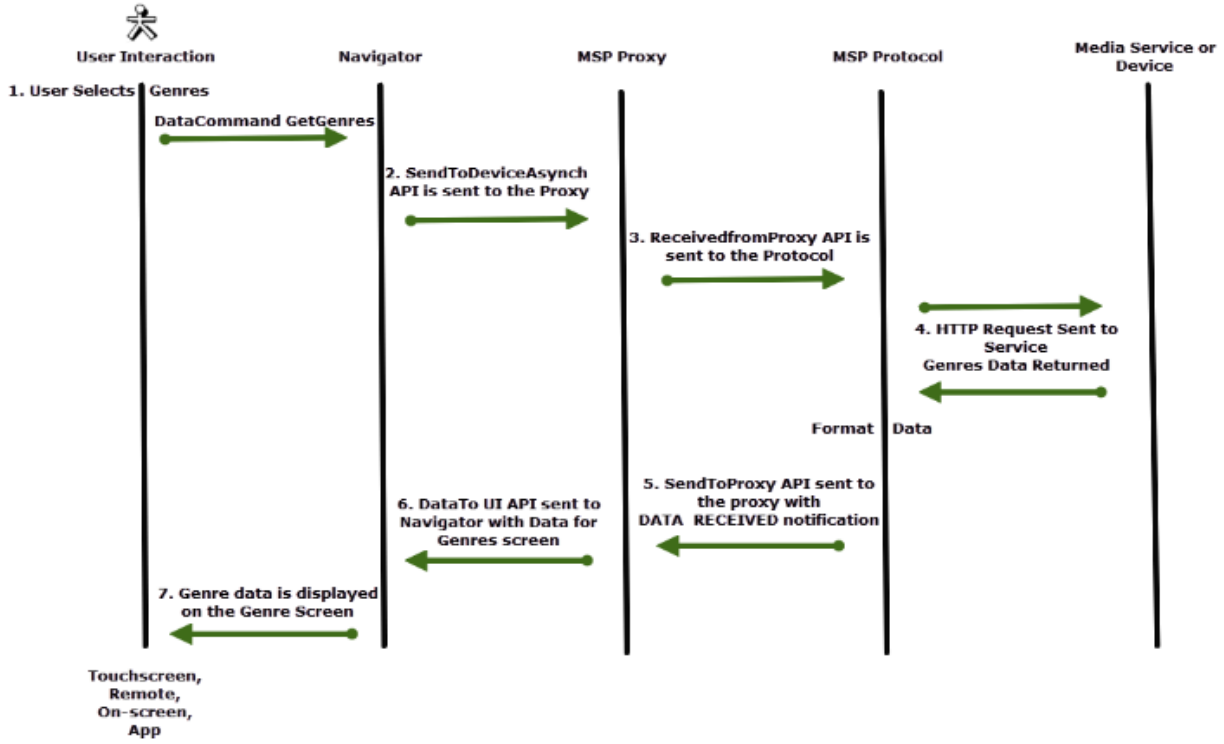
The Hello World driver was created as a driver to demonstrate the capabilities of the Media Service Proxy through a driver created for a media streaming service. Driver created for device using the MSP are somewhat different. In addition, the Hello World driver, two additional sample drivers are included to explain the XML and Lua differences between drivers for devices and streaming services. They are called HW Selected Device and HW Streaming Device. These two drivers are discussed in detail in the Chapter titled: Comparing Device drivers with Streaming Service Drivers.

Overview of the Media Service Proxy & Driver Development

Before we begin discussing the details behind the various driver components that move data and screens through the Media Service Proxy, reviewing the diagram below will offer a visual overview of the process. We will refer to the stages represented in this diagram throughout this document.

In the diagram below the use case example we'll use is that of a user interacting with a media service. The user has pressed the Listen icon from Navigator and selected the Media Service. The diagram begins at the point where the user selects a "Genres" tab from the media Service's screen:

Data Flow through the Media Services Proxy



1. The user initiates a request through a user interface. This may be a MyHome app, touchscreen, onscreen navigator or even List Navigator on a System Remote. The specific request is to access a list of music genres from the media service. When the button is pressed on the UI, it evokes a DataCommand called GetGenres

2. The Media Service Proxy receives the Genres screen request through a Control4 API called SendToDeviceAsync. The API has several parameters. These include:
   - The ProxyDevice ID
   - The GetGenres data command
   - The ID of the Navigator device where the request originated
   - A Sequence value that is used to associate this request withe the data that will be coming back from the media service.

- The Room ID where the request originated
- ARGS parameter which consists of a table of parameters with UI Information.
- A Local parameter which is an optional string parameter that represents the geographic locale of the Navigator device. A prototype of the command would look like this:

```
SendToDeviceAsync(ProxyDeviceID, GetGeneres, ARGS, NAVID, SEQ, ROOMID, LOCALE)
```

3. After receiving the API from Navigator, the proxy sends the GetGenres command to the protocol using the ReceivedfromProxy API. The API has the following parameters:

- The idBinding
- The GetGenres command
- A table of parameters which include the ARGS, NAVID, SEQ, ROOMID and LOCALE as defined in Step 2. A prototype of the command would look like this:

```
ReceivedFromProxy(idBinding, GetGenres, tParams(ARGS, NAVID, SEQ, ROOMID, LOCALE))
```

4. The protocol received the GetGenres command and its parameters and then sends an HTTP requests for the Genre data from the media service. The media service returns the data to the protocol. The protocol is responsible for formatting the returned data to a spec which is supported by the MSP.

5. Now, the protocol needs to send the formatted Genres data to the proxy. It does this using the SendToProxy API with the DATA_RECEIVED protocol notification. The API has a set of parameters which include:

The idBinding value
The DATA_RECEIVED protocol notification

A table of parameters that includes the formatted genres DATA, an error handling parameter, a NAVID and a SEQ value which must match the NAVID and SEQ values passed in the previous steps. A prototype of the command would look like this:

```
SendToProxy(idBinding, "DATA_RECEIVED", tParams(DATA, ERROR, NAVID, SEQ))
```

6. Now that the proxy has the genres data and its associated parameters, it will use the DataToUI API to send the data to Navigator. The API has the following parameters;

- The ProxyDeviceID
- The NAVID and SEQ values defined in previous steps.
- The Data for the Genres screen
- An Error handling parameter

A prototype of the command would look like this:

```
DataToUI(ProxyDeviceID, NavID, SEQ, DATA, ERROR)
```

7. Navigator delivers the data to populate the Genres Screen. It matches the data properties to display on the screen using the following convention:

```
<List>
        <item>
                <genre>Pop</genre>
        </item>
</List>
```

The Purpose of a Media Service Driver

Drivers represent the framework of a Control4 Project. Any device or service that requires control subsequently requires a driver. Drivers developed using Media Service Proxy represent a unique opportunity to provide efficient data delivery combined with a customized user experience.

The driver you develop will be specific to a media device or media service and it will interact with several architectural layers and need to handle all aspects of device or service control. It will not only use just MSP, but also the protocol layer which is unique to the device or service as well as Navigator. Your driver will depend on the defined Media Service Schema document to render screens to serve up data as well as respond to user requests.

The Role of Navigator

Navigator is the Control4 component that facilitates user interaction with a device. Navigator runs on devices where user interaction is supported. Devices such as touchscreens, mobile devices, and remote controls all run instances of Navigator. The Media Service Proxy views Navigator as the end point for data that will be displayed to the end user. The MSP provides a layer between the UI displayed through Navigator and the data coming from the media device or service. The Media Service Proxy receives the data from the device (or protocol driver), manipulates that data in a manner that Navigator can consume it – and then delivers it to Navigator for display using the DriverWorks API OnDatatoUI.

What makes the Media Service Proxy unique is its ability to deliver not just informational data, but UI elements as well. Control4 has defined a schema for the Media Service Proxy that contains UI elements that can be customized and delivered to Navigator for display to the end user.

The Role of the Protocol

When we refer to the "Protocol" in this document we are referring to driver notifications that originate at the device, media service or the driver itself and are sent to the Media Service Proxy. The MSP receives this data using the SendToProxy API and then acts accordingly. It may simply pass the data on to Navigator for display on the UI. It may store the protocol data in a queue and wait for more data to arrive before doing anything at all.

When the MSP receives data from the Protocol, it also dictates what UI elements are needed to support the appropriate display of the protocol data on Navigator devices. The schema defined in this part of the driver code can dictate what type of screen will be displayed to present the data, how the data is organized on the screen as well as deliver elements that can capture user interaction such as buttons and links to other areas on the UI.

The data sent from the Protocol is delivered through a set of defined Protocol Notifications. These are outlined in the Appendix under: MSP Protocol Notification section.

The Protocol also receives data from the Media Service Proxy in the form of Proxy Commands. This is data that originates at the UI level, move through the Media Service Proxy for parsing and then is passed to the Protocol. This data is sent using the

ReceivedFromProxy API. A list of supported Commands is outlined in the Appendix under: Media Service Proxy Commands.

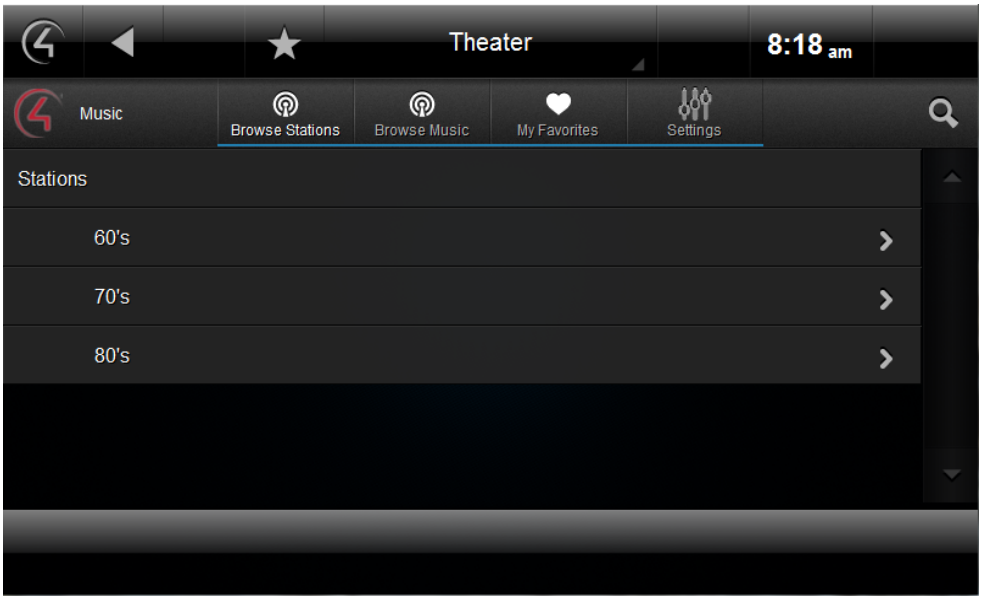
Understanding Screens and the Media Service Proxy

In the terms of this document, a screen represents the visual representation of driver data that is delivered through the Media Service Proxy. The proxy supports implementations on supported devices such as iPad, an Android tablet, Control4 Touchscreens, etc. It is also important to consider orientations of portrait and landscape with respect to each device. A key feature of the proxy is its ability to support the display of a consistent look and feel while simultaneously optimizing UI elements for an individual device.

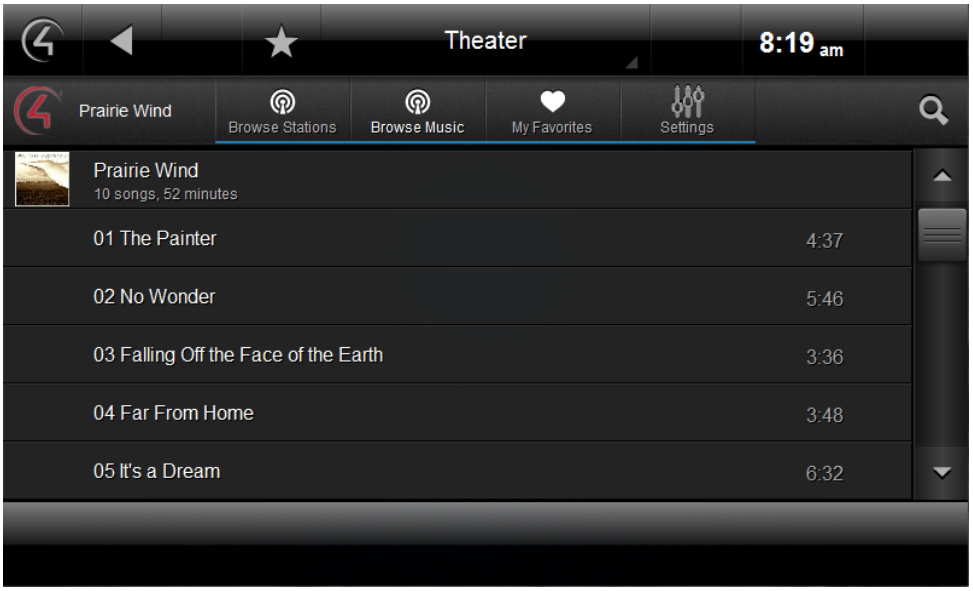
For example, the number of rows displayed on a screen varies according to the available screen real estate. Another example is the number of icons displayed on a certain area of the UI. The Media Service Proxy has the ability to show the optimal number of elements based on device and orientation without driver development efforts for each interface.

There are three types of screens supported through the Media Service Proxy: a List Screen, a Collection Screen and a Settings Screen. These Screen Types are the pre-defined UI frameworks which define the layout of a screen. However, for some devices it may not be applicable to adhere to the suggested layout – especially if the layout is not appropriate for that device.

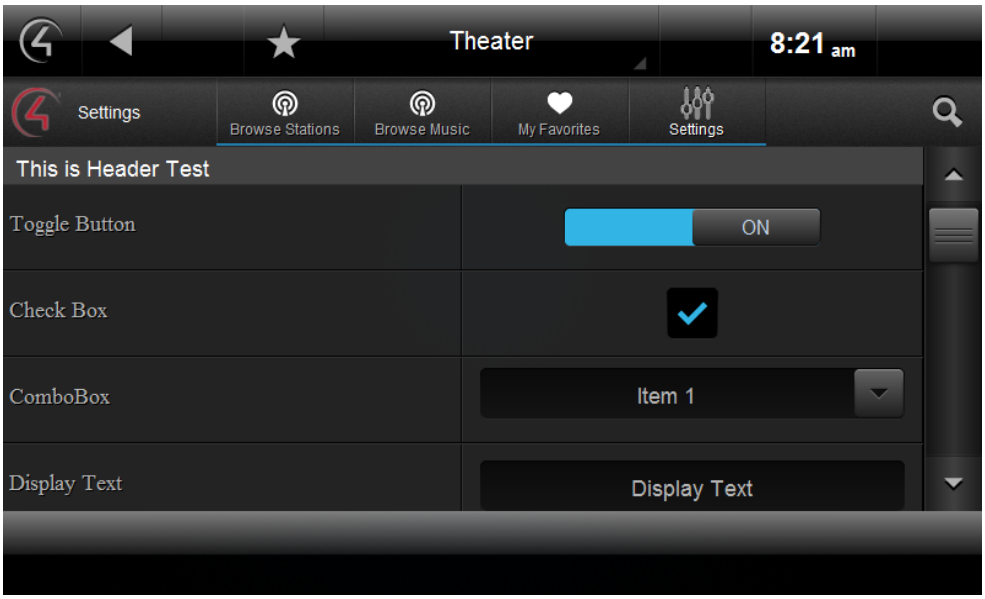
Assigning a Type to a screen determines the screen layout. For example, the image below is a List screen from the sample Hello World driver:



Here is an example of a Collection Type Screen:



Here is an example of the Settings Type Screen:



The sections of the document that follow will explain how each of the three screen types were developed in the Hello World driver in conjunction with the Media Service Proxy. Before reviewing this content, you may find it helpful to have several resources configured. This will allow you to get the most of the information and examples that follow. These resources include:

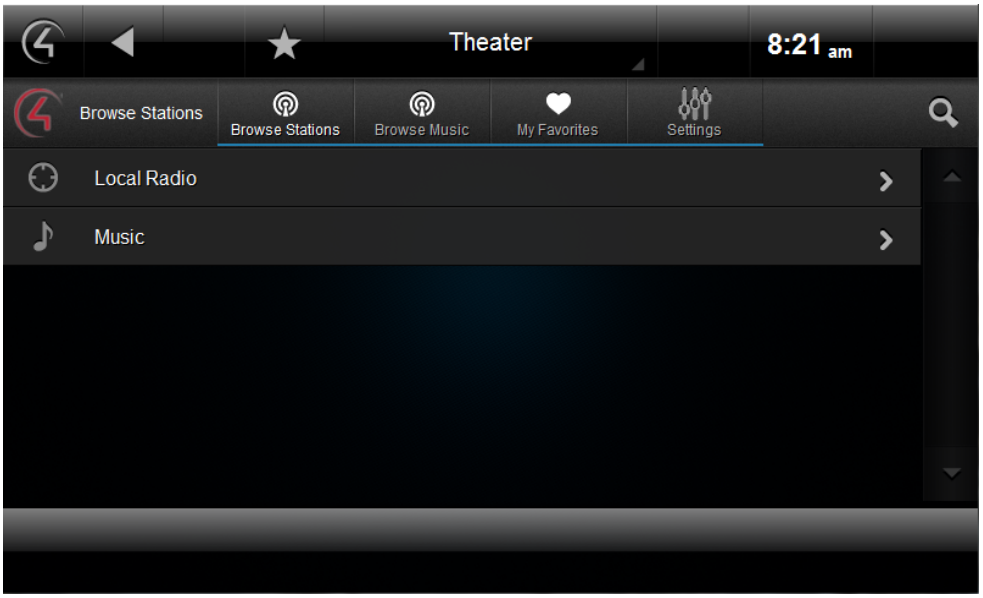
1. A Control4 Project which is running 2.6.0 for both ComposerPro and Director
2. The included Hello World driver should be loaded into the project.

3. A device running Navigator or MyHome connected to Director for viewing the user interface
4. A text editing tool of your choice such as NotePad++ or UltraEdit. The examples in this document use FirstObject.

Building Screens and Handling Driver Data

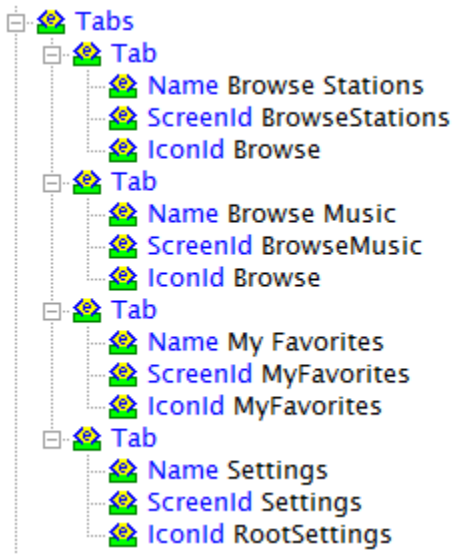
Building a List Type Screen

As indicated by its name, a List screen displays un-related data in a list format. The List screen type is useful as it presents data in an organized manner to the end user. This section will review the code used to display the List screen after a user selects the Hello World driver from the Listen menu in Navigator. Specifically, the screen shown below:



Before we examine the code behind the list objects displayed on the screen (Local Radio and Music), let's look at the top of the screen and see how the tabs are displayed. The tabs we're referring to are named: "Browse Stations", "Browse Music", "My Favorites" and "Settings".

Using an XML editor, open the Hello World driver and navigate to the "**capabilities -> UI-> Tabs"** section. It looks like this:

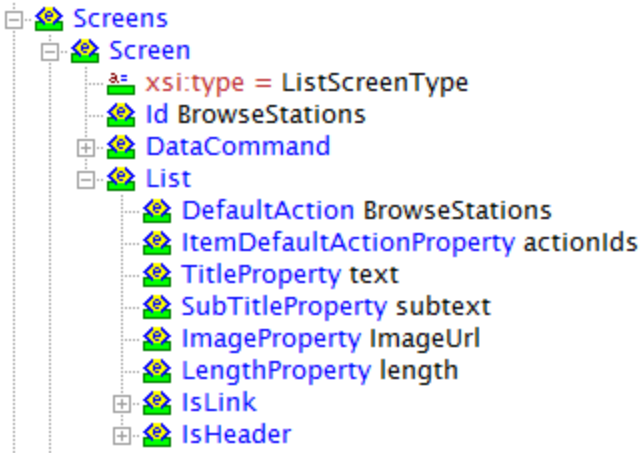


We can see that this driver has four defined tabs. These are the tabs displayed across the top of our screen. Tabs are rather simple in that they have three elements:

The Name value is the name of the tab and what the user will see on the UI. The Screen ID value is how the screen will be identified by the Media Service Proxy and referred to in any proxy commands. Finally, the icon value is the name of the image that is displayed on the tab itself. If we look at the driver code for the Browse Station Tab, it looks like this:

```
<Tab>
        <Name>Browse Stations</Name>
        <ScreenId>BrowseStations</ScreenId>
        <IconId>Browse</IconId>
</Tab>
```

When a user selects this tab, the MSP will refer to the ScreenId value and load that screen, or in this case the Browse Stations screen. Now is good time to look at the driver xml code for the screen definitions. From your XML editor, open the capabilities -> UI->Screens. Open the first screen in the list. It will look like this:



Here is where our Browse Stations screen is defined. You can see the ID value of BrowseStation. This is the same value passed for a screen ID in our Tab definition.

As mentioned, the Browse Stations screen is a List Type screen. Next, we'll examine the code to see how this screen is defined. The first half of the code sample we'll review is below:

```
<Screen xsi:type="ListScreenType">
                              <Id>BrowseStations</Id>
                              <DataCommand>
                                       <Name>GetBrowseStationsMenu</Name>
                                       <Type>PROTOCOL</Type>
                                       <Params>
                                             <Param>
                                                    <Name>screen</Name>
                                                    <Type>DEFAULT</Type>
                                                    <Value>BrowseStations</Value>
                                             </Param>
                                             <Param>
                                                    <Name>URL</Name>
                                                    <Type>FIRST_SELECTED</Type>
                                                    <Value>URL</Value>
                                             </Param>
                                             <Param>
                                                    <Name>key</Name>
                                                    <Type>FIRST_SELECTED</Type>
                                                    <Value>key</Value>
                                                    </Param>
                                             <Param>
                                                    <Name>search</Name>
                                                    <Type>SEARCH</Type>
                                             </Param>
                                             <Param>
                                                    <Name>search_filter</Name>
                                                    <Type>SEARCH_FILTER</Type>
                                             </Param>
                                       </Params>
                              </DataCommand>
```

The first line in the sample code designates this screen as a List screen type. Next we can see the screen's ID value again: "BrowseStations." As mentioned earlier, this is the ID referenced in the Browse Station tab. This ID value is how the MSP knows what screen to load on to the UI.

Next, we can see the beginning of a DataCommand definition. When a screen loads, the first thing it does is fire a DataCommand. The purpose of the DataCommand is to provide the protocol driver a definition to use to build the initial list of data which is being requested from the UI. The name of the DataCommand in this screen is called GetBrowseStationsMenu. We can see it defined between the <name> </name> tags. Immediately following the name element we can see a command Type. It is defined as PROTOCOL, meaning that this data command is meant to go from the UI, through the MSP to the protocol driver.

After the DataCommand definition we see the parameters for the command. Parameters are also referred to as ARGs in the MSP. It's important to understand the different types of parameters, or ARGS that are defined in the parameters section of the Browse Stations screen code. If you look at the five parameters defined for this screen you'll see the following parameter types: `DEFAULT`, `FIRST_SELECTED`, `SEARCH`, and `SEARCH_FILTER`.

If a parameter has a type of `DEFAULT`, that parameter will always be sent with the DataCommand. For example, here is the first parameter defined for the DataCommand:

```
<Param>
        <Name>screen</Name>
        <Type>DEFAULT</Type>
        <Value>BrowseStations</Value>
</Param>
```

The parameter's name is screen, its type is default and its value is BrowseStations. This means that anytime the GetBrowseStationsMenu DataCommand is sent, it will include this parameter named "screen" with a value of "BrowseStation." This is the same value as the Screen ID as defined in the Tab and subsequently our Tab name.

The next two parameters have a type of `FIRST_SELECTED`. Parameters with this type are only sent when the user makes a list selection from the UI. Since displaying the Browse Station screen didn't involve a user selection from a list, we'll move past these parameters for now. They will be discussed later in this section.

The next two parameters types: `SEARCH` and `SEARCH_FILTER` are involved with the search functionality of the driver. Search values are what the user enters into the UI keyboard when attempting to retrieve specific data. Note that both `SEARCH` and `SEARCH_FILTER` values are always returned in a search attempt. Like `FIRST_SELECTED`, neither of these parameters are used in this specific example, but they will be discussed in detail in an upcoming section.

Now that we've seen how the DataCommand portion of our screen is defined, let's exercise that functionality on Navigator and simultaneously review the debug output in ComposerPro to see how it works. When the Browse Station tab is selected on the UI, the following output is displayed:

```
ReceivedFromProxy(): GetBrowseStationsMenu on binding 5001; Call Function GetBrowseStationsMenu()
ROOMID:  7
ARGS:  <args><arg name="screen">BrowseStations</arg></args>
SEQ:  3
NAVID:  11
Browsing main menu
GetBrowseMenu (5001, 3) for nav 11
SendToProxy (5001, DATA_RECEIVED)
```

The top of the output shows us our DataCommand "GetBrowseStationMenu" being sent through the Control4 API ReceivedfromProxy. This command is going from the UI to the protocol. We can see several other elements being sent. These include ROOMID, SEQ and NAVID. These are always passed and their inclusion is handled by the Media Service Proxy Schema for you. You will see these parameters being passed often through the MSP. They are defined within the MSP Command section in the appendix as well as here:

NAVID – String parameter that represents a Navigator Device ID. The device id is used so that Navigator can filter out replies from other Navigators devices. Some Navigator devices might not have a driver. In that case a unique id will be generated.

SEQ - Integer parameter that is a Navigator-generated sequence ID value. The ID value is useful when trying to match correct replies with Navigator requests.

ROOMID - Integer parameter that is the ID value of the room where the Navigator is located

You'll notice another element in the debug list called ARGS. ARGS or arguments handles any parameters associated with the command. If we look at the debug we see that the ARGS has a name of "screen" and a value of "BrowseStations." ARGS handles any of the parameters we've defined in our DataCommand. We've already discussed that the only parameter being sent in the command is the default parameter with a name of "screen" and a value of "BrowseStations." That's why we see the debug line of:

```
ARGS: <args><arg name="screen">BrowseStations</arg></args>
```

To understand how the ARGS parameters are handled, we need to look at the how the GetBrowseStationMenu function is defined in the lua portion of the Hello World driver:

```
function PRX_CMD.GetBrowseStationsMenu(idBinding, tParams)
	print("GetBrowseStationsMenu (" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
tParams.NAVID)
	local args = ParseProxyCommandArgs(tParams)

	local tListItems = {}
	local url
	local isRootMenu = false
	local screen = args.screen
	local key = args.key
	local search = args.search
	if (search ~= nil) then
		local searchFilter = args.search_filter
		if ((searchFilter == "station") or (searchFilter == "show")) then
			local tTemp = g_browse_localradio
			for i,v in pairs(tTemp) do
				if (string.match(string.upper(v.text), string.upper(search))) then
					table.insert(tListItems, v)
				end
			end
		elseif ((searchFilter == "song") or (searchFilter == "artist")) then

		end
		print("Searching (" .. searchFilter .. ") for " .. search)
	else
		if (key == "radio") then
			tListItems = g_browse_localradio
		elseif (key == "music")       then
			tListItems = g_browse_stations
		elseif (key == "music_60s")   then
			tListItems = g_browse_stations_60s
		elseif (key == "music_70s")   then
			tListItems = g_browse_stations_70s
		elseif (key == "music_80s")   then
			tListItems = g_browse_stations_80s
		else
			tListItems = g_browse_mainmenu
		end

		if (key == nil) then
			key = "main menu"
		elseif  (key == "music") then
			--the music option will build another browse screen
		else
			--add actionIds for the ItemDefaultActionProperty
			for j,k in pairs(tListItems) do
				k["actionIds"] = "PlayStations"
			end
		end

		print("Browsing " .. key)

	end
```

You'll notice the use of a helper function called ParseProxyCommandArgs in this function. This helper function creates a table of parameters from the block of XML code which is sent by the proxy. A table format is much more suitable when working in Lua. That's why it is defined as: `tParams = ParseProxyCommandArgs(tParams)`

What we see in our Lua Output window for the GetBrowseStationMenu command are the table elements of tParams. They are arranged in a Key = Value format from the table. For example, when we see the output window display: ROOMID: 57, it is printing a Key of ROOMID with its paired Value of 57.

When the ARGS data is sent from the Proxy it is one long XML string. Our example only has one parameter so it's fairly simple. However, if we were looking at another command that had numerous parameters using `FIRST_SELECTED`, `SEARCH` or `SEARCH_TYPE`, parsing the xml into a table makes the data easier to work with.

It's important to understand that all of the parameters for our DataCommand will be wrapped in the <args></args> tags when they are sent in ReceivedfromProxy. The reason why <arg> parameters need to be handled separately from the other elements is that an individual arg can be nested with numerous other ones. For example, if we needed to get the Room ID we could get that value with a simple statement such as: `tParams["ROOMID"]` That becomes more difficult when we have to traverse a lengthy XML string. Here is where `tParams = ParseProxyCommandArgs(tParams)` comes into play.

A table of parameters (tParams) comes down from the Proxy and it is passed into the GetBrowseStationMenu here:

```
function PRX_CMD.GetBrowseStationsMenu(idBinding, tParams)
```

That tParams table is then sent as a parameter to the `ParseProxyCommandArgs` helper function. `ParseProxyCommandArgs` then parses the XML block and creates a table with the all the name value pairs that that it contained. This ARG table is then appended to the tParams table. The modified tParams table is then returned. To further understand how this works, let's look at the `ParseProxyCommandArgs` function. Here it is from our sample driver:

```
function ParseProxyCommandArgs(tParams)
	local args = {}
	local parsedArgs = C4:ParseXml(tParams["ARGS"])
	for i,v in pairs(parsedArgs.ChildNodes) do
		args[v.Attributes["name"]] = v.Value
	end
	return args

end
```

We can see that tParams is passed as a parameter. Then a local table called args is created. Next, a Control4 API called ParseXML is called and passed an "ARGS" parameter. This is the entire XML string of arg parameters from the DataCommand. The API takes that string and returns it as an object. The object is iterated through in pairs based on the name of the arg and value of the arg. For example, "screen" and "BrowseStations". This entry is then made in the args table. This is only applicable to the args parameters. The other parameters that are handles by the MSP schema are not modified at all. Elements such as ROOMID, SEQ and NAVID are not impacted by this.

Parsing the XML parameters of the DataCommand into a lua table in this manner allows us to easily pull out individual parameters (or args) much easier than having to traverse through a potentially lengthy XML string every time data was needed. For example, now that our XML string is formatted into a .Lua table, we can get the value of the `SEARCH_FILTER` parameter with:

```
local searchFilter =  tParams.ARGS.search_filter
```

In the above example, we can pull the data out of the arg in the XML by referring to the tParams table at the .ARGS record and then reference the search filter data element.

Up until this point, we've looked at how data is sent from the Navigator through user interaction, through proxy and to the protocol. Remember, the data request originated with the selection of the Browse Stations tab. Now, let's look at how the protocol driver will build the screen that we want to display when the tab is selected as well as how the MSP handles the data coming from the protocol driver up to the display.

If we refer to the Data Flow through the Media Service Proxy diagram, we can see that data is returned from the protocol using the helper function DATA_RECEIVED with the SendToProxy commands. Here is an example from our Hello World driver:

```
function DataReceived(idBinding, navId, seq, response)
	local data
	if (type(response) == "table") then
		data = BuildListXml(response, false)
	else
		data = response
	end

	local tResponse = {
		["NAVID"] = navId,
		["SEQ"] = seq,
		["DATA"] = data,
	}
	SendToProxy(idBinding, "DATA_RECEIVED", tResponse)
end
```

We can see a local table created called tResponse. Besides the data to build the screen, this table also has a NAVID element that ensure that the data being sent to the correct device, a SEQ number that will match the SEQ value sent from the proxy at the point of the request and the ROOMID value. These are the same values that came down from the proxy in the ReceivedFromProxy command. This helper function receives the response parameter in one of two manners: if the data is received from the proxy in the form of a table, it then uses the BuildListXml function to assemble the table contents. If the data is received in a string format, it passes the string data as is.

If we look at the definition of the DATA_RECEIVED – we can see that one of its parameters above is called "DATA". That parameter requires an XML list of the data that we want to display on the screen. In the example above, we can see the use of another helper function (BuildListXml) to build the list with the proper XML syntax. It is passed a table (t). We defined the table t in our Hello World driver as a table called g_browse_main menu or:

```
t = g_browse_mainmenu
```

If we look at that table in our driver, we'll see this:

```
g_browse_mainmenu = {
{type = "link", folder = "true", text = "Local Radio", URL =
"http://opml.radiotime.com/Browse.ashx?c=local&formats=mp3,aac,wma&partnerId=7vGvIDO5&serial=000F
FF501D75", key = "radio", image = "ico_tunein_localradio.png"},

{type = "link", folder = "true", text = "Music", URL =
"http://opml.radiotime.com/Browse.ashx?c=music&formats=mp3,aac,wma&partnerId=7vGvIDO5&serial=000F
FF501D75", key = "music", image = "ico_tunein_music.png"},
```

g_browse_mainmenu is a table that has the parameters we need to build a list to display on the screen. You can see the text value of Local Radio and Music in the code above. They have a URL and an image. Notice the Key value. In the first example Key is equal to "radio". This is used in the definition of the GetBrowseStations command:

```
function PRX_CMD.GetBrowseStationsMenu(idBinding, tParams)
	print("GetBrowseMenu (" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
tParams.NAVID)
	local args = ParseProxyCommandArgs(tParams)

	local tListItems = {}
	local url
	local isRootMenu = false
	local screen = args.screen
	local key = args.key
	local search = args.search
	if (search ~= nil) then
		local searchFilter = args.search_filter
		if ((searchFilter == "station") or (searchFilter == "show")) then
			local tTemp = g_browse_localradio
			for i,v in pairs(tTemp) do
				if (string.match(string.upper(v.text), string.upper(search))) then
					table.insert(tListItems, v)
				end
			end
		elseif ((searchFilter == "song") or (searchFilter == "artist")) then

		end
		print("Searching (" .. searchFilter .. ") for " .. search)
	else
		if (key == "radio") then
			tListItems = g_browse_localradio
		elseif (key == "music")		then
			tListItems = g_browse_stations
		elseif (key == "music_60s")	then
			tListItems = g_browse_stations_60s
		elseif (key == "music_70s")	then
			tListItems = g_browse_stations_70s
		elseif (key == "music_80s")	then
			tListItems = g_browse_stations_80s
		else
			tListItems = g_browse_mainmenu
		end
		if (key == nil) then key = "main menu" end
		print("Browsing " .. key)
	end
```

We can see in the example above that if the key is equal to radio the driver gets the g_browse_localradio table or:

```
	if (key == "radio") then
	   t = g_browse_localradio
```

Similarly, if the key is equal to music, as in the second part of our g_browse_mainmenu table, our driver will get the g_browse_stations table or:

```
	elseif (key == "music")		then
```

```
        t = g_browse_stations
```

If we review the debug code we can see how the BuildListXml helper function really works. When our screen loads the screen data from the SendToProxy(idBinding, "DATA_RECEIVED", tParams) our Lua output window looks like this:

```
ReceivedFromProxy(): GetBrowseStationsMenu on binding 5002; Call Function GetBrowseStationsMenu()
ROOMID: 7
ARGS: <args><arg name="screen">BrowseStations</arg></args>
SEQ: 33
NAVID: 11
GetBrowseMenu (5002, 33) for nav 11
Browsing main menu
SendToProxy (5002, DATA_RECEIVED)
SEQ: 33
NAVID: 11
DATA: <List><item><folder>true</folder><type>link</type><key>radio</key><URL>http://opml.radiotime.com/Browse.ashx?
c=local&formats=mp3,aac,wma&partnerId=7vGvIDO5&serial=000FFF501D75</URL><text>Local Radio</text><image>ico_tunein_localradio.png
</image></item><item><folder>true</folder><type>link</type><key>music</key><URL>http://opml.radiotime.com/Browse.ashx?
c=music&formats=mp3,aac,wma&partnerId=7vGvIDO5&serial=000FFF501D75</URL><text>Music</text><image>ico_tunein_music.png</image>
</item></List>
ARGS: table: 0x43be4550
screen: BrowseStations
ROOMID: 7
```

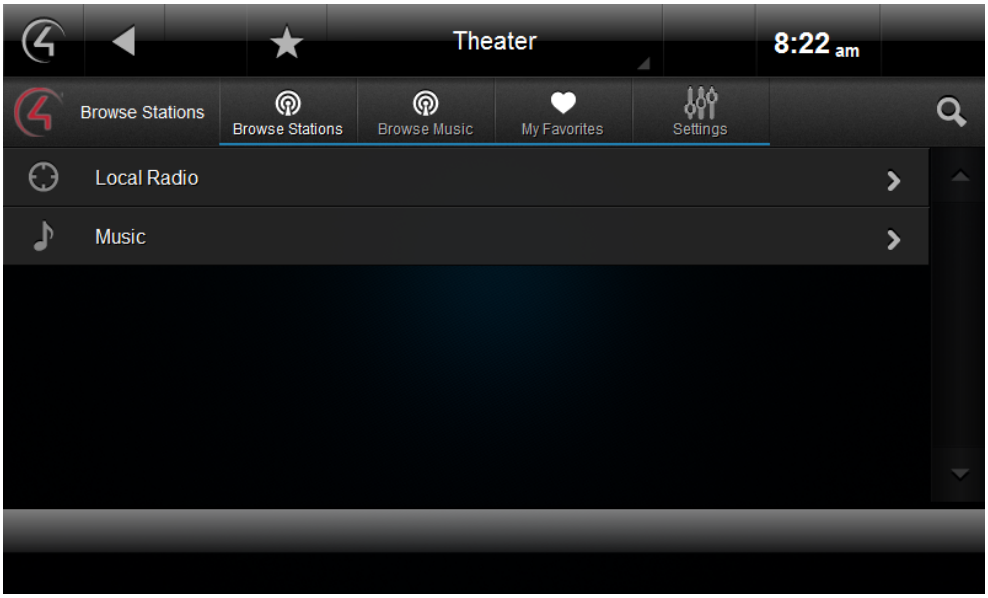
First we see our GetBrowseStationsMenu command and the MSP parameters such as ROOMID and SEQ as well as our DataCommand parameters or ARGS. Notice the DATA section. This is the string of XML built into a list of Key Value pairs by the BuildListXml helper function. The first part of the code is our Local Radio data and the second is the Music. If we looked at the DATA content in XML it would look like this:

```
<List>
        <item>
                <folder>true</folder>
                <type>link</type>
                <key>radio</key>
                <URL>URL>
                <text>Local Radio</text>
                <image>ico_tunein_localradio.png</image>
        </item>
        <item>
                <folder>true</folder>
                <type>link</type>
                <key>music</key>
                <URL></URL>
                <text>Music</text>
                <image>ico_tunein_music.png</image>
        </item>
</List>
```

BuidListXml takes each of the table entries of g_browse_mainmenu and wraps them in an <item> tag within a list. BuildListXml is defined in our driver as follows:

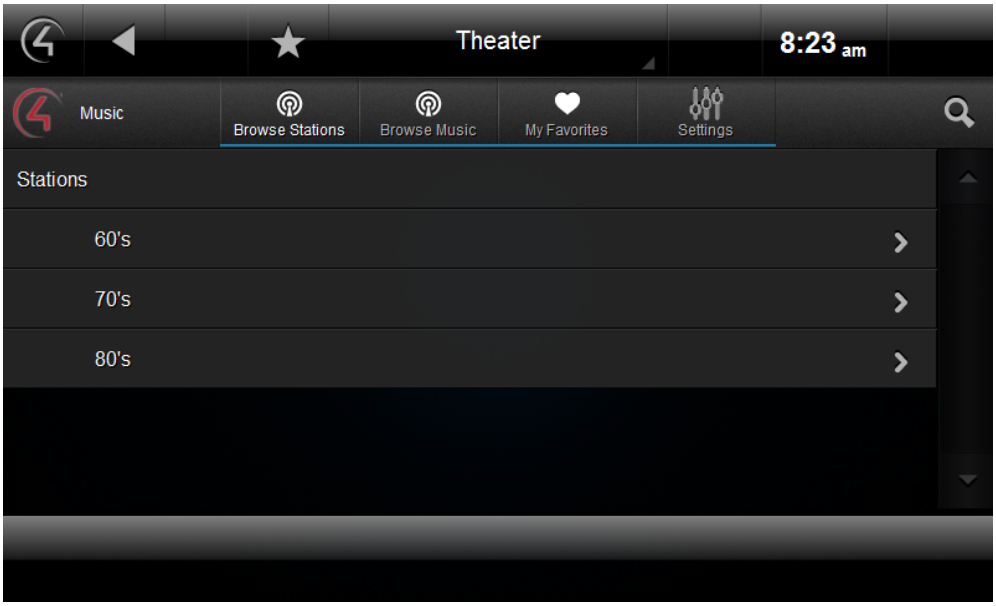
```
function BuildListXml(tData, escapeValue)
	local xml = ""

	xml = "<List>"

	if (escapeValue) then
		for j,k in pairs(tData) do
			xml = xml .. "<item>"
			for i,v in pairs(k) do
				xml = xml .. "<" .. i .. ">" .. C4:XmlEscapeString(v) .. "</" .. i
				.. ">"
			end
			xml = xml .. "</item>"
		end
	else
		for j,k in pairs(tData) do
			xml = xml .. "<item>"
			for i,v in pairs(k) do
				xml = xml .. "<" .. i .. ">" .. v .. "</" .. i .. ">"
			end
			xml = xml .. "</item>"
		end
	end

	xml = xml .. "</List>"

	return xml
end
```

The Data manifests itself in our display as this:

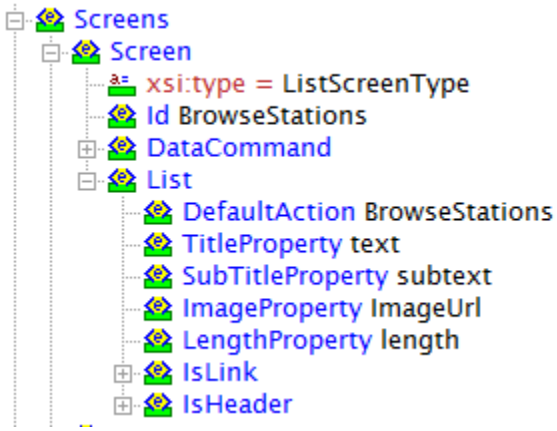


Now that we have the Browse Station list displayed, the user can make selection from the list: Local Radio or Music. When this happens, our driver builds and displays another list in a similar manner as it did when the list was built and displayed from selecting the Browse Station tab. However, there are some differences.

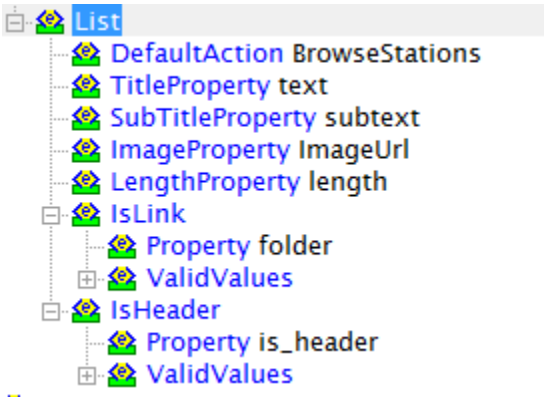
If we select the Music option from our list the following screen is displayed:



Now is good time to go back and look at the rest of the driver code for Browse Station screen. From your text editor, open the **"capabilities -> UI->Screens."** Open the first screen in the list. Once again, this is where our Browse Stations screen is defined:



Up until now we've been reviewing the code and data involved with the first half of the code for this screen. Here is the second half:

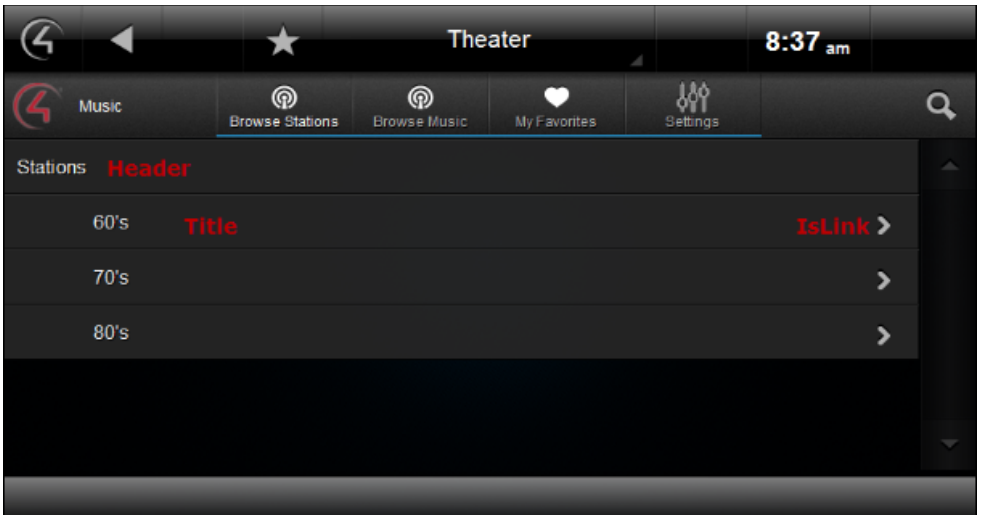


The first item to review in the above code sample is the Default Action. Default Action occurs inside the List element and differs from the DataCommand. The Default Action references a single action that is run if no other actions are defined or enabled. You'll notice that it has parameters very similar to the DataCommand. After the Default Action is defined you'll see some other elements that are defined by the MSP schema. These include: Title Property, SubTitle Property, Image Property, Length Property, IsLink, and IsHeader. The value for each can be called whatever you like. In our sample driver we have called them as follows:

Title Property = text
SubTitle Property = subtext
Image Property = image
Length Property = length
IsLink = folder
IsHeader = Is_Header

Definitions for additional MSP schema elements can be found in the schema document.

However these schema elements are named, that's how we'll refer to them in the screen. Some of these elements are displayed on our Hello World screen as follows:



Keep in mind, this screen was displayed because of the Default Action being fired. The name of that command is BrowseStationsCommand. That command is sent to the protocol with the list of parameters as defined in our code ample above.

We can easily confirm this by selecting the Music option in the Browse Station main menu and reviewing the lua output in ComposerPro:

```
ReceivedFromProxy(): GetBrowseStationsMenu on binding 5001; Call Function GetBrowseStationsMenu()
ROOMID: 6
ARGS: <args><arg name="screen">BrowseStations</arg></args>
SEQ: 8
NAVID: 10
GetBrowseMenu (5001, 8) for nav 10
SendToProxy (5001, DATA_RECEIVED)
Browsing main menu
SEQ: 8
NAVID: 10
DATA: <List><item><folder>true</folder><type>link</type><key>radio</key><URL></URL><text>Local Radio</text>
<ImageUrl>ico_tunein_localradio.png</ImageUrl></item><item><folder>true</folder><type>link</type><key>music</key>
<URL></URL><text>Music</text><ImageUrl>ico_tunein_music.png</ImageUrl></item></List>
```

We can see ReceivedfromProxy used once again and it is sending the BrowseStationsCommand and its defined parameters and ARG data. It is worthwhile reviewing the way the args are defined for the BrowseStationsCommand. The first arg for screen uses a default parameter type just as we used it in the DataCommand. Here is the key arg:

```
<Param>
        <Name>key</Name>
        <Type>FIRST_SELECTED</Type>
        <Value>key</Value>
</Param>
```

Notice how the <Name> of the arg is key and the <Value> is also a key. An arg can be named anything you want. However, you'll likely find it useful to keep the Name and Value entries identical as the name used here will be how this arg will be displayed in our debug in ComposerPro. But more importantly, it is how this parameter will be referenced in the driver's code. This value is the parameter in the existing list from which the MSP gets its data and subsequently sends to the protocol to populate the UI. Once again here is the Lua output from executing the default action. You can see all of the args that are defined in the command definition displayed in the window:

```
ReceivedFromProxy(): GetBrowseStationsMenu on binding 5001; Call Function GetBrowseStationsMenu()
ROOMID: 6
ARGS: <args><arg name="screen">BrowseStations</arg></args>
SEQ: 8
NAVID: 10
GetBrowseMenu (5001, 8) for nav 10
SendToProxy (5001, DATA_RECEIVED)
Browsing main menu
SEQ: 8
NAVID: 10
DATA: <List><item><folder>true</folder><type>link</type><key>radio</key><URL></URL><text>Local Radio</text>
<ImageUrl>ico_tunein_localradio.png</ImageUrl></item><item><folder>true</folder><type>link</type><key>music</key>
<URL></URL><text>Music</text><ImageUrl>ico_tunein_music.png</ImageUrl></item></List>
```

One important aspect to understand when looking at the args content in the lua ouput window above is that we first passed this data when the GetBrowseStations DataCommand was originally fired. This is also the same data we discussed in the BuildListXml example. That was loaded when the main menu was displayed and when the user pressed Music from the list, data that has the "music" key value is displayed. Now is good time to look at the BrowseStationCommand function and how it is defined:

```
function PRX_CMD.BrowseStationsCommand(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)
	local tResponse = {}
	local nextscreen

	if (args.type == "link") then
		nextscreen = "<NextScreen>BrowseStations</NextScreen>"
		DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)

	elseif (args.type == "audio") then
		--clear NowPlaying  if song is playing
		if (gNowPlaying[1] ~= nil) then
			if (gNowPlaying[1].type == "song") then
				gNowPlaying = {}
			end
		end

		--Add Song info to gNowPlaying Queue
		table.insert(gNowPlaying, g_MediaByKey[args.key])
		gNowPlaying[#gNowPlaying].Id = #gNowPlaying
		gCurrentSongIndex = #gNowPlaying
		local id = gCurrentSongIndex

		--***Select the Song to Play ***
		SelectInternetRadio(idBinding, tParams["ROOMID"], gNowPlaying[id].URL, id)

		--***UpdateMediaInfoForRoom***
		UpdateMediaInfo(idBinding, gNowPlaying[id].Title, gNowPlaying[id].SubTitle, "Line
		3", "Line 4", gNowPlaying[id].ImageUrl, tParams["ROOMID"], "secondary", "True")

		--***Goto Now Playing screen***
		nextscreen = "<NextScreen>#nowplaying</NextScreen>"
		DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)


	else
		print("PRX_CMD.BrowseStationsCommand(), unhandled 'type' parameter:" .. args.type)
		return
	end
```

When a user selects and item from the List, we can see that this function parses the arguments into a table. Then we can see several arguments have been created. Specifically:

```
	if (args.type == "link") then
		nextscreen = "<NextScreen>BrowseStations</NextScreen>"
		DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)

	elseif (args.type == "audio") then
		--clear NowPlaying  if song is playing
		if (gNowPlaying[1] ~= nil) then
			if (gNowPlaying[1].type == "song") then
				gNowPlaying = {}
			end
		end
```

Depending on what the user selects on the screen, a type `(record_type)` is associated with that selection. Based on the way the BrowseStationsCommand is defined, if the type is a link type, the command builds XML called nextscreen and it calls the same screen – Browse Stations. This is all done with the same list we used to load the Browse Station Main Menu– the Browse Stations List. This XML (nextscreen with the BrowseStation) is placed in the DATA parameter table. This table gets sent down to the protocol and is returned through the SendToProxy API using the DataReceived helper function, or:

```
SendToProxy(idBinding, "DATA_RECEIVED", tParams)
```

This is the same model we used in moving data with our Default Action. However, this was initiated by a user selection from a List and subsequently, the BrowseStation Command is fired with all of its ARG data. Because these ARGS include a key = to "music" it will load the data found in the g_browse_stations table. We can see this defined in the GetBrowseStationMenu command:

```
function PRX_CMD.GetBrowseStationsMenu(idBinding, tParams)
	print("GetBrowseMenu (" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
tParams.NAVID)
	local args = ParseProxyCommandArgs(tParams)

	local tListItems = {}
	local url
	local isRootMenu = false
	local screen = args.screen
	local key = args.key
	local search = args.search
	if (search ~= nil) then
		local searchFilter = args.search_filter
		if ((searchFilter == "station") or (searchFilter == "show")) then
			local tTemp = g_browse_localradio
			for i,v in pairs(tTemp) do
				if (string.match(string.upper(v.text), string.upper(search))) then
					table.insert(tListItems, v)
				end
			end
		elseif ((searchFilter == "song") or (searchFilter == "artist")) then

		end
		print("Searching (" .. searchFilter .. ") for " .. search)
	else
		if (key == "radio") then
			tListItems = g_browse_localradio
		elseif (key == "music")       then
			tListItems = g_browse_stations
		elseif (key == "music_60s")   then
			tListItems = g_browse_stations_60s
		elseif (key == "music_70s")   then
			tListItems = g_browse_stations_70s
		elseif (key == "music_80s")   then
			tListItems = g_browse_stations_80s
		else
			tListItems = g_browse_mainmenu
		end
		if (key == nil) then key = "main menu" end
		print("Browsing " .. key)
	end

	DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], tListItems)

end
```

Specifically, this line:

```
        elseif (key == "music")       then
                        t = g_browse_stations
```

When we reviewed this command earlier we didn't have a key because the command was called from pressing the Browse Stations tab. That time it loaded the g_browse_mainmenu table. This time it is called from a user selection "Music" from the list. So, the g_browse_station table is loaded. Here is the g_browse_station table from the Hello World driver:

```
g_browse_stations = {
        {text="Stations", is_header="true"},
        {type = "link", folder = "true", text = "60's", URL = "", key = "music_60s"},
        {type = "link", folder = "true", text = "70's", URL = "", key = "music_70s"},
        {type = "link", folder = "true", text = "80's", URL = "", key = "music_80s"},
```

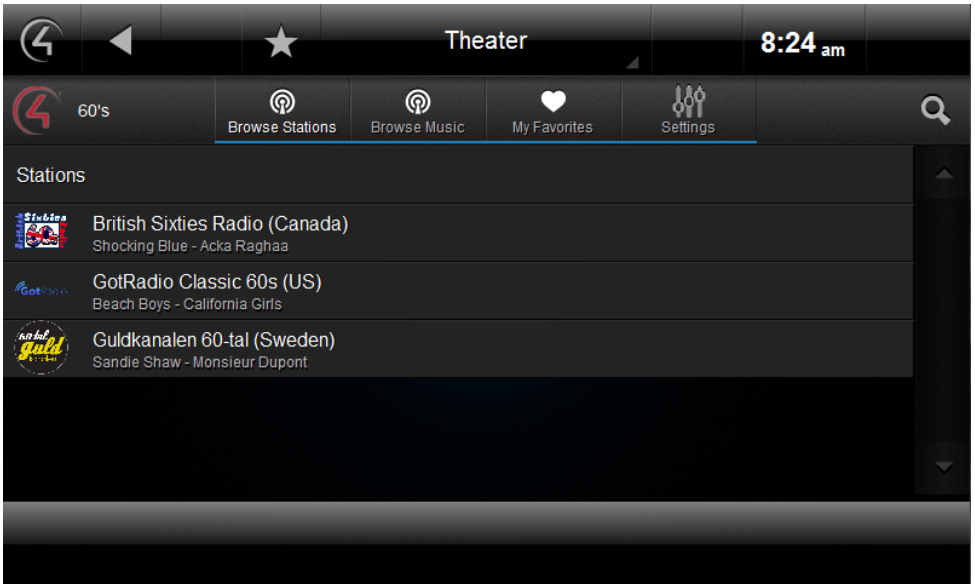
This lists all of the stations on our UI. If a user selected "60's" from that list, the GetBrowseStationsmenu command handles that request with:

```
        elseif (key == "music_60s")   then
                        t = g_browse_stations_60s
```

The key on that selection is "music_60s". The driver then loads the g_browse_stations_60s table and displays the table data on the UI. Here is the g_browse_stations 60s table:

```
g_browse_stations_60s = {
        {text="Stations", is_header="true"},
        {type = "audio", folder = "false", text = "British Sixties Radio (Canada)", subtext =
        "Shocking Blue - Acka Raghaa", URL = "", key = "ShockingBlue", ImageUrl = gMediaPath ..
        "Stations/BritishSixtiesRadio.png"},
        {type = "audio", folder = "false", text = "GotRadio Classic 60s (US)", subtext = "Beach
        Boys - California Girls", URL = "", key = "BeachBoys", ImageUrl = gMediaPath ..
        "Stations/GotRadio.png"},
        {type = "audio", folder = "false", text = "Guldkanalen 60-tal (Sweden)", subtext =
        "Sandie Shaw - Monsieur Dupont", URL = "", key = "SandieShaw", ImageUrl = gMediaPath ..
        "Stations/Guldkanalen.png"},
}
```

Here is the resulting UI:



Refreshing Screens
The RequiresRefresh XML tag can be included in the screen's XML to refresh the screen by resending the original DataCommand for a normal screen or by sending a GetQueue for the now playing screen. This response is NOT supported on commands executed by notifications.

```
<RequiresRefresh>true</RequiresRefresh>
```

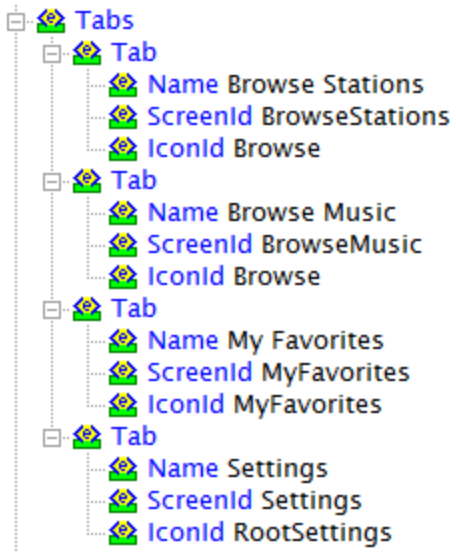
Building a Collection Type Screen

Note: Several of the development concepts used in this section are detailed in the prior chapter called Building a List Type Screen. It is recommended that the Building a List Type Screen section be reviewed and its concepts understood before continuing on to the Collection type Screen development content.

A Collection screen is similar to a List screen in that it also displays data in a list format. However, it differs in that all of the data displayed on a Collection screen is identical in some manner and usually represents a group of like data. A good example of a Collection Screen's content would be songs contained within an album.

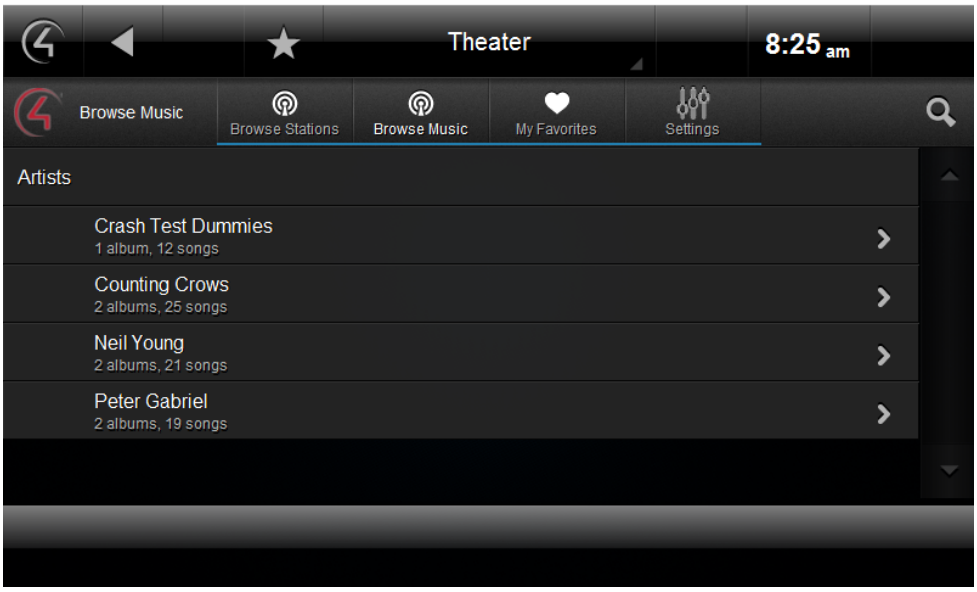
In the previous section, we reviewed how a List Screen type is used in our Hello World driver to support the display of a list of stations and music. Now that we understand how that screen works, let's see how a Collection Screen type is used to display groups of like data – such as albums from an artist or songs within an album.

When we began reviewing the List screen content in the previous section we started with the premise that the user would access the data from within the UI by selecting a tab. Our example used the Browse Stations tab. As a review, here is the XML from our driver which supports the tabs across the top of the screen in our UI:



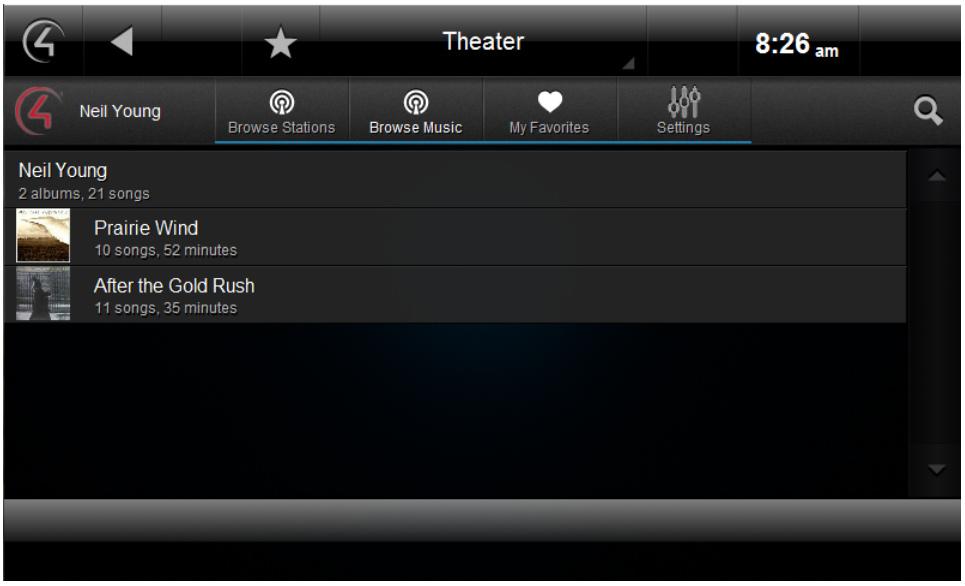
This concept still holds true for a Collection type screen. However, in our Hello World driver the user accesses the Collection type screen by first selecting a List screen. Keep in mind that a Collection can just as easily be accessed through a Tab as well.

The usage scenario we'll address in this section begins with a user selecting the Browse Music tab on the Hello World driver's UI. This action returns an initial List type screen which contains a list of artists, the number of albums available from the artist as well as the total number of songs. To see this screen, select the Browse Music tab from the UI:



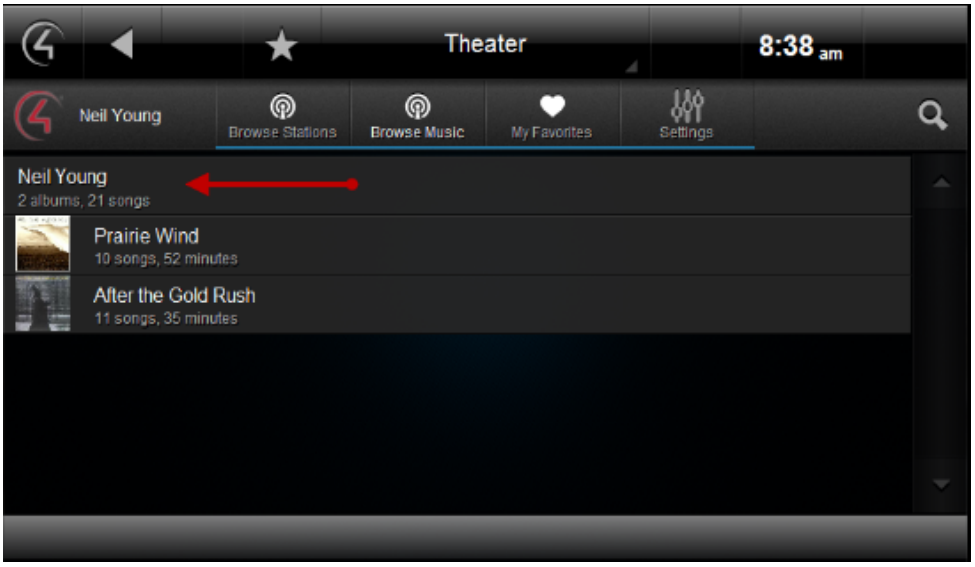
Here we can see another List screen displayed. This is provided in the same manner as outlined in the Building a List Type Screen section.

The List screen displayed presents the user with a decision to select an artist. Our user is a Neil Young fan and when Neil Young is selected, the data returned is delivered using a Collection Type screen:



Now, let's look at the Hello World debug code in ComposerPro and the driver code to understand how this data is provided through the MSP.

When our user selects Neil Young from the list on the UI, our debug output in ComposerPro looks like this:



We can see that the BrowseMusicCommand function is called. If we look at how that command is defined in our driver's lua code, we see this:

```
function PRX_CMD.BrowseMusicCommand(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)
	if (args.type == "artist") then
		local nextscreen = "<NextScreen>BrowseMusicCollection</NextScreen>"
		DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)
	else
		print("PRX_CMD.BrowseMusicCommand(), unhandled 'type' parameter:" .. args.type)
		return
	end

end
```

Remember, our user selected an "artist" or Neil Young. By selecting an artist, the BrowseMusicCommand, which is the default action for our List screen displaying artists, uses the <NextScreen> element. The <NextScreen> element which loads the BrowseMusicCollection screen.

As we discussed in the List screen section, when a screen is loaded its DataCommand is the first thing that is executed. If we look at how the BrowseMusicCollection screen is defined in the driver's xml, we see this:

```
<Screen xsi:type="CollectionScreenType">
                              <Id>BrowseMusicCollection</Id>
                              <DataCommand>
                                     <Name>GetBrowseMusicCollectionMenu</Name>
                                     <Type>PROTOCOL</Type>
                                     <Params>
                                            <Param>
                                                   <Name>start</Name>
                                                   <Type>DATA_OFFSET</Type>
                                            </Param>
                                            <Param>
                                                   <Name>count</Name>
                                                   <Type>DATA_COUNT</Type>
                                            </Param>
                                            <Param>
                                                   <Name>key</Name>
                                                   <Type>FIRST_SELECTED</Type>
                                                   <Value>key</Value>
                                            </Param>
                                            <Param>
                                                   <Name>type</Name>
                                                   <Type>FIRST_SELECTED</Type>
                                                   <Value>type</Value>
                                            </Param>
                                     </Params>
                              </DataCommand>
```

The DataCommand executed is called GetBrowseMusicCollectionMenu. Let's look at how that DataCommand is defined in our driver's lua code:

```
function PRX_CMD.GetBrowseMusicCollectionMenu(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)

	local collection, list, data
	data = ""
	local tCollectionItems = {}

	local search = args.search
	if (search ~= nil) then
		local searchFilter = args.search_filter
		if (searchFilter == "song") then
			for i,v in pairs(g_music_albums) do
				for j,k in pairs(g_music_SongsByAlbum[i]) do
					if (string.match(string.upper(k.name),
					string.upper(search))) then
						collection = BuildSimpleXml("Collection",v, false)
						local tTemp = {}
						table.insert(tTemp, k)
						list = BuildListXml(tTemp, false)
						break
					end
				end
			end
			data = collection .. list
			print("Searching (" .. searchFilter .. ") for " .. search)
		else
			print("Searching (" .. searchFilter .. ") for " .. search)
			return
		end

	else
		if (args.type == "artist") then

			for j,k in pairs(g_music_artists) do
				if (j == args.key) then
					collection = BuildSimpleXml("Collection", k, false)
					data = collection
					for i,v in pairs(g_music_albums) do
						if (k.name == v.artist) then
							table.insert(tCollectionItems,v)
						end
					end
					list = BuildListXml(tCollectionItems, false)
					data = collection .. list
					break
				end
			end
		elseif (args.type == "album") then
			collection = BuildSimpleXml("Collection",g_music_albums[args.key], false)
			list = BuildListXml(g_music_SongsByAlbum[args.key], false)
			data = collection .. list
		else
			collection = BuildSimpleXml("Collection",g_music_albums[args.key], false)
			list = BuildListXml(g_music_SongsByAlbum[gNowPlaying[#gNowPlaying].key],
		false)
			data = collection .. list
		end
	end

	DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], data)

end
```

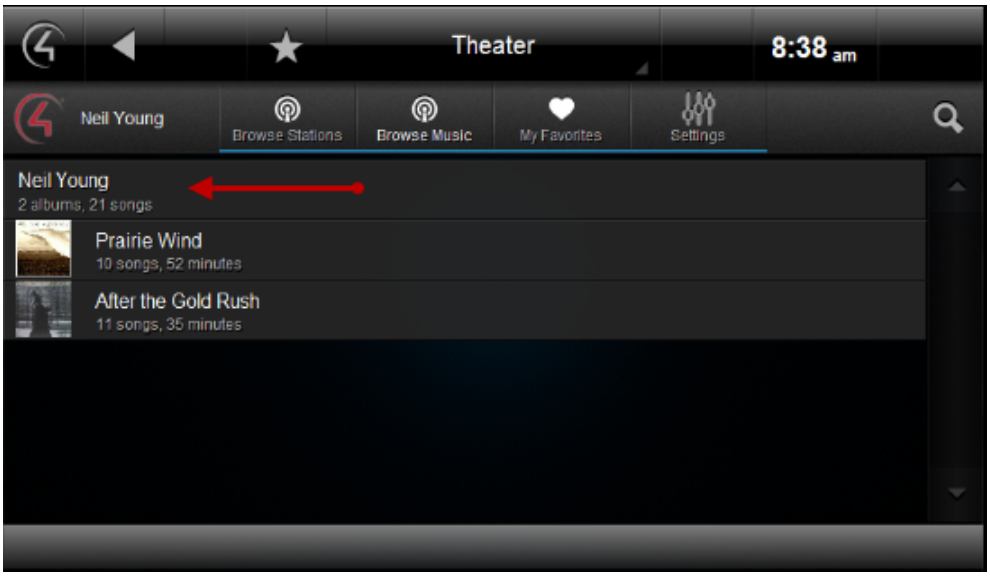
Again, this is the DataCommand for the Collection screen. It uses many of the concepts we detailed in the Building a List Screen Type section. We can see at the beginning that the usual parameters are passed with this command such as ROOMID, NAVID, LOCALE and so on. As we saw in our List Screen Type section, we are once again parsing the ARGs elements and placing them in a table of parameters. This forms our data into a consistent table of Key Value pairs from the XML.

Next, the command has two arguments: ("type" and "key") or: is the selection an "artist" or an "album"? Our user selected Neil Young or an "artist" which created a "type" argument with the value "artist". So, the driver looks through the data in the g_music_artists table and looks for a match for "Neil Young". When the user presses Neil Young on the screen, it created an argument "key" with the value "neil_young". You can see this key created in the output window of ComposerPro when that selection is made.

Now, let's look at the g_music_artist table:

```
g_music_artists = {
        ["Crash Test Dummies"] = {type = "artist", name = "Crash Test Dummies", info = "1 album,
        12 songs"},
        ["Counting Crows"] = {type = "artist", name = "Counting Crows", info = "2 albums, 25
        songs"},
        ["Neil Young"] = {type = "artist", name = "Neil Young", info = "2 albums, 21 songs"},
        ["Peter Gabriel"] = {type = "artist", name = "Peter Gabriel", info = "2 albums, 19
        songs"},
}
```

The last entry in the table is Neil Young. This where the data shown on the UI in the header area as seen below:



Now that the GetBrowseMusicCollection command has found "Neil Young" in the g_music_artists table, it uses the BuildSimpleXML helper command to assemble that data into "Collection" This is then passed into a local variable named "data."

Next, the command iterates through the g_music_albums table. It will find any albums that match the artist. If we look at the g_music_albums table, we see this:

```
g_music_albums = {
        ["God Shuffled His Feet"] = {type = "album", artist = "Crash Test Dummies", name = "God
        Shuffled His Feet", info = "12 songs, 45 minutes", key = "God Shuffled His Feet",
        ImageUrl = gMediaPath .. "Crash Test Dummies/God Shuffled His Feet/Folder.jpg",},
        ["Recovering the Satellites"] = {type = "album", artist = "Counting Crows", name =
        "Recovering the Satellites",info = "14 songs, 59 minutes", key = "Recovering the
        Satellites", ImageUrl = gMediaPath .. "Counting Crows/Recovering the
        Satellites/Folder.jpg",},
        ["August and Everything After"] = {type = "album", artist = "Counting Crows", name =
        "August and Everything After",info = "11 songs, 52 minutes", key = "August and Everything
        After", ImageUrl = gMediaPath .. "Counting Crows/August and Everything
        After/Folder.jpg",},
        ["Prairie Wind"] = {type = "album", artist = "Neil Young", name = "Prairie Wind", info =
        "10 songs, 52 minutes", key = "Prairie Wind", ImageUrl = gMediaPath .. "Neil
        Young/Prairie Wind/Folder.jpg",},
        ["After the Gold Rush"] = {type = "album", artist = "Neil Young", name = "After the Gold
        Rush", info = "11 songs, 35 minutes", key = "After the Gold Rush", ImageUrl = gMediaPath
        "Neil Young/After the Gold Rush/Folder.jpg",},
        ["Up"] = {type = "album", artist = "Peter Gabriel", name = "Up",info = "11 songs, 77
        minutes", key = "Up", ImageUrl = gMediaPath .. "Peter Gabriel/Up/Folder.jpg",},
        ["So"] = {type = "album", artist = "Peter Gabriel", name = "So",info = "11 songs, 77
        minutes", key = "So", ImageUrl = gMediaPath .. "Peter Gabriel/So/Folder.jpg",},
}
```

The table contains two albums with an "artist" value of "Neil Young": Prairie Wind and After the Gold Rush. The command pulls the data for those two albums and uses the BuildListXML helper function to create an XML list of that data. It then updates the final "DATA" parameter to include both collection and list content.

See the line in the GetBrowseMusicCollectionMenu: `tParams["DATA"] = data`

Let's take a moment to review what we've just covered. Our user has selected Neil Young from the Browse music tab. This loaded a collection type screen and executed its DataCommand called GetBrowseMusicCollectionMenu. The command knows it's been passed a key named "neil_young". It found the artist in the g_music_artist table. It then took the appropriate data and built a collection of it.

We can see this returned on our ComposerPro output when the user selects Neil Young from the screen. The DATA output looks like this:

```
DATA:  <Collection><type>artist</type><name>Neil Young</name><info>2 albums, 21
songs</info></Collection><List><item><type>album</type><name>Prairie Wind</name><key>Prairie
Wind</key><artist>Neil Young</artist><info>10 songs, 52
minutes</info></item><item><type>album</type><name>After the Gold Rush</name><key>After the Gold
Rush</key><artist>Neil Young</artist><info>11 songs, 35 minutes</info></item></List>
```

If we take the <Collection> part of the output and arrange it in XML it would look like this:

```
<Collection>
        <type>artist</type>
        <name>Neil Young</name>
        <info>2 albums, 21 songs</info>
</Collection>
```

This data defines the collection shown on the UI screen. We declared a local variable called "data" and placed the collection list (created by BuildSimpleXML) into it.

See the line in the GetBrowseMusicCollectionMenu: `data = collection .. list`

Next, the command used the artist value of neil_young to iterate the g_music_albums table. When any entries with an artist name equal to neil_young was found, the command inserted that data into a table. BuildListXML was used to build a local variable named "list" with this information. We can see this in the second half of the Data output in ComposerPro:

```
DATA:  <Collection><type>artist</type><name>Neil Young</name><info>2 albums, 21
songs</info></Collection><List><item><type>album</type><name>Prairie Wind</name><key>Prairie
Wind</key><artist>Neil Young</artist><info>10 songs, 52
minutes</info></item><item><type>album</type><name>After the Gold Rush</name><key>After the Gold
Rush</key><artist>Neil Young</artist><info>11 songs, 35 minutes</info></item></List>
```

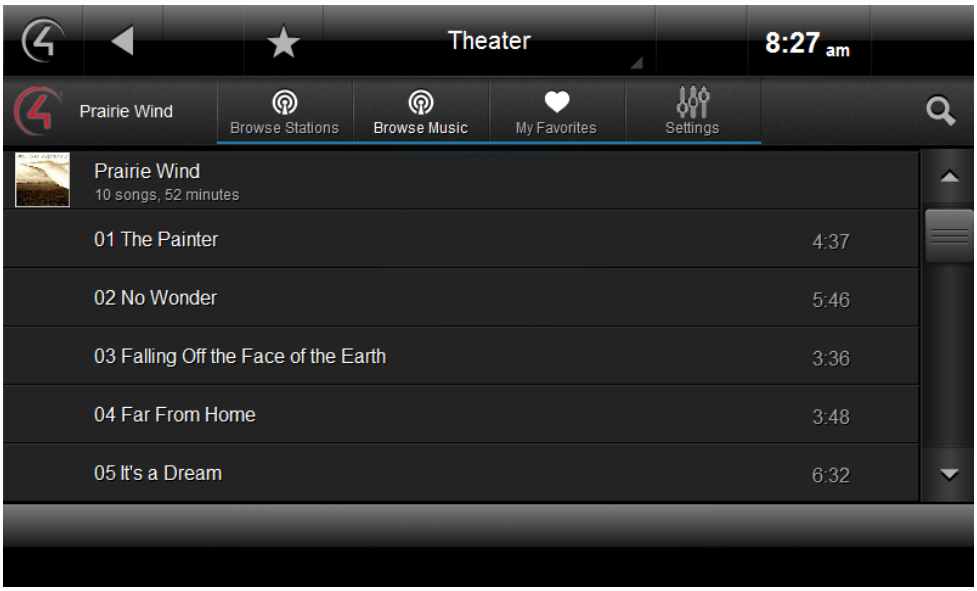
If we take the <List> part of the output and arrange it in XML it would look like this:

```
<List>
        <item>
                <type>album</type>
                <name>Prairie Wind</name>
                <key>Prairie Wind</key>
                <artist>Neil Young</artist>
                <info>10 songs, 52 minutes</info>
        </item>

        <item>
                <type>album</type>
                <name>After the Gold Rush</name>
                <key>After the Gold Rush</key>
                <artist>Neil Young</artist>
                <info>11 songs, 35 minutes</info>
        </item>
</List>
```

This list information is added to our "data" variable along with the collection content. This data then populates the Collection screen on the UI.

See the line in the GetBrowseMusicCollectionMenu: `data = collection .. list`

Now, let's say our user wants to listen to Neil Young's Prairie Wind album. Our Hello World driver builds yet another collection screen to present songs by album information so it can be displayed on the UI as in the example below:



If we look at the lua output in ComposerPro to see what happens when a user selects the Prairie Wind album we'll see this:

```
ReceivedFromProxy(): BrowseMusicCollectionCommand on binding 5002; Call Function
BrowseMusicCollectionCommand()
ROOMID:  7
ARGS:  <args><arg name="screen">BrowseMusicCollection</arg><arg name="type">album</arg><arg
name="key">Prairie Wind</arg><arg name="type">album</arg></args>
SEQ:  75
NAVID:  11
SendToProxy (5002, DATA_RECEIVED)
SEQ:  75
NAVID:  11
ROOMID:  7
DATA:  <NextScreen>BrowseMusicCollection</NextScreen>
ARGS:  table: 0x43c10b78
type:  album
screen:  BrowseMusicCollection
key:  Prairie Wind
ReceivedFromProxy(): GetBrowseMusicCollectionMenu on binding 5002; Call Function
GetBrowseMusicCollectionMenu()
ROOMID:  7
ARGS:  <args><arg name="start">0</arg><arg name="count">20</arg><arg name="key">Prairie
Wind</arg><arg name="type">album</arg></args>
SEQ:  76
SendToProxy (5002, DATA_RECEIVED)
SEQ:  76
NAVID:  11
NAVID:  11
ROOMID:  7
DATA:  <Collection><type>album</type><name>Prairie Wind</name><key>Prairie Wind</key><artist>Neil
Young</artist><info>10 songs, 52 minutes</info></Collection><List><item><key>The
Painter</key><type>song</type><name>The
Painter</name><playbackSeconds>3:35</playbackSeconds></item><item><key>No
Wonder</key><type>song</type><name>No
Wonder</name><playbackSeconds>3:39</playbackSeconds></item><item><key>Falling Off the Face of the
Earth</key><type>song</type><name>Falling Off the Face of the
Earth</name><playbackSeconds>3:51</playbackSeconds></item><item><key>Far From
Home</key><type>song</type><name>Far From
Home</name><playbackSeconds>3:35</playbackSeconds></item><item><key>It's a
Dream</key><type>song</type><name>It's a
Dream</name><playbackSeconds>3:35</playbackSeconds></item><item><key>Prairie
Wind</key><type>song</type><name>Prairie
Wind</name><playbackSeconds>3:35</playbackSeconds></item><item><key>Here for
You</key><type>song</type><name>Here for
You</name><playbackSeconds>3:35</playbackSeconds></item><item><key>This Old
Guitar</key><type>song</type><name>This Old
Guitar</name><playbackSeconds>3:35</playbackSeconds></item><item><key>He Was the
King</key><type>song</type><name>He Was the
King</name><playbackSeconds>3:35</playbackSeconds></item><item><key>When God Made
Me</key><type>song</type><name>When God Made
Me</name><playbackSeconds>3:35</playbackSeconds></item></List>
ARGS:  table: 0x43c4c490
start:  0
type:  album
count:  20
key:  Prairie Wind
```

In our previous collection examples – we received collections of "neil_young". Now we are receiving a collection of "prarie_wind". This is because the user clicked on a list item on the collection screen called "Prairie Wind." When that happens, the DefaultAction for the Browse Music Collection screen is executed. Again, this is different than our previous example which initially loaded the Browse Music Collection screen and fired its DataCommand.

The DefaultAction executed is called BrowseMusicCollectionCommand. It can be seen here in our driver XML where the screen is defined:

```
<DefaultAction>
        <Name>BrowseMusicCollectionCommand</Name>
        <Type>PROTOCOL</Type>
        <Params>
                <Param>
                        <Name>screen</Name>
                        <Type>DEFAULT</Type>
                        <Value>BrowseMusicCollection</Value>
                </Param>
                <Param>
                        <Name>type</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>type</Value>
                </Param>
                <Param>
                        <Name>URL</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>URL</Value>
                </Param>
                <Param>
                        <Name>item</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>item</Value>
                </Param>
                <Param>
                        <Name>is_preset</Name>
                        <Type>FIRST_SELECTED</Type>
                <Value>is_preset</Value>
                </Param>
                <Param>
                        <Name>key</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>key</Value>
                </Param>
                <Param>
                        <Name>type</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>type</Value>
                </Param>
                <Param>
                        <Name>text</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>text</Value>
                </Param>
                <Param>
                        <Name>image</Name>
                        <Type>FIRST_SELECTED</Type>
                        <Value>image</Value>
                </Param>
        </Params>
</DefaultAction>
```

It can also be seen in the first line of our output window when Prairie Wind is selected by the user. Let's look at how the command is defined in our Hello World driver:

```
function PRX_CMD.BrowseMusicCollectionCommand(idBinding, tParams)
        local args = ParseProxyCommandArgs(tParams)
        if (args.type == "album") then
                local nextscreen = "<NextScreen>PlayMusicCollection</NextScreen>"
                DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)
        else
                print("PRX_CMD.BrowseMusicCollectionCommand(), unhandled type: " .. args.type)
        end

end
```

We can see that it also has an argument for "type". If the type selected is "album" then it will call the PlayMusicCollection screen. We can see this between the `<NextScreen>` tags. This is the same screen we called in our first example however it gets called this time using a different "type." Previously, we called the screen with the type of "artist" with the value "Neil Young." Now we are using the type of album with the value "Prairie Wind." This can be seen in the ARGS output in our ComposerPro window:

```
ARGS:  <args><arg name="screen">BrowseMusicCollection</arg><arg name="type">album</arg><arg
name="key">Prairie Wind</arg><arg name="type">album</arg></args>
```

As in previous examples, when we call a screen we execute its DataCommand. The DataCommand for the PlayMusicCollection screen is called GetPlayMusicCollectionMenu:

```
<Screen xsi:type="CollectionScreenType">
	<Id>PlayMusicCollection</Id>
	<DataCommand>
		<Name>GetPlayMusicCollectionMenu</Name>
		<Type>PROTOCOL</Type>
		<Params>
			<Param>
				<Name>start</Name>
				<Type>DATA_OFFSET</Type>
			</Param>
			<Param>
				<Name>count</Name>
				<Type>DATA_COUNT</Type>
			</Param>
			<Param>
				<Name>key</Name>
				<Type>FIRST_SELECTED</Type>
				<Value>key</Value>
			</Param>
			<Param>
				<Name>type</Name>
				<Type>FIRST_SELECTED</Type>
				<Value>type</Value>
			</Param>
			<Param>
				<Name>search</Name>
				<Type>SEARCH</Type>
			</Param>
			<Param>
				<Name>search_filter</Name>
				<Type>SEARCH_FILTER</Type>
			</Param>
		</Params>
	</DataCommand>
```

Let's look at how that command is defined in our driver's lua code:

```
function PRX_CMD.GetPlayMusicCollectionMenu(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)

	local collection, list, data
	collection = BuildSimpleXml("Collection",g_music_albums[args.key], false)
	list = BuildListXml(g_music_SongsByAlbum[args.key], false)
	data = collection .. list
	DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], data)

end
```

Notice the key that is highlighted in the code sample above. Specifically, this is the key value for Prairie Wind. This can be seen in our lua output:

```
ARGS:  <args><arg name="screen">PlayMusicCollection</arg><arg name="type">album</arg><arg
name="key">Prairie Wind</arg><arg name="type">album</arg></args>
```

Based on how our command is defined, if a type of album is passed, the command looks at the g_music_albums table. The table looks like this:

```
g_music_SongsByAlbum = {
        ["God Shuffled His Feet"] = {type = "album", artist = "Crash Test Dummies", name = "God
        Shuffled His Feet", info = "12 songs, 45 minutes", key = "God Shuffled His Feet",},
        ["Recovering the Satellites"] = {type = "album", artist = "Counting Crows", name =
        "Recovering the Satellites",info = "14 songs, 59 minutes", key = "Recovering the
        Satellites",},
        ["August and Everything After"] = {type = "album", artist = "Counting Crows", name =
        "August and Everything After",info = "11 songs, 52 minutes", key = "August and Everything
        After",},
        ["Prairie Wind"] = {type = "album", artist = "Neil Young", name = "Prairie Wind", info =
        "10 songs, 52 minutes", key = "Prairie Wind",},
        ["After the Gold Rush"] = {type = "album", artist = "Neil Young", name = "After the Gold
        Rush", info = "11 songs, 35 minutes", key = "After the Gold Rush"},
        ["Up"] = {type = "album", artist = "Peter Gabriel", name = "Up",info = "11 songs, 77
        minutes", key = "Up",},
        ["So"] = {type = "album", artist = "Peter Gabriel", name = "So",info = "11 songs, 77
        minutes", key = "So",},
        }
```

If it finds a table containing the key value of "Prairie Wind" it will pull that data. BuildSimpleXml takes that data and formats into "collection". Next it iterates through the g_music_SongsByAlbum table looking for a match to the key value which is Prairie Wind. This table is lengthy and can be seen in in its entirety in the sample driver. It does contain and entry with the value Prairie Wind. The entry looks like this:

```
["Prairie Wind"] = {
                {type = "song", name="01 The Painter", playbackSeconds="4:37", key="The Painter"},
                {type = "song", name="02 No Wonder", playbackSeconds="5:46", key="No Wonder"},
                {type = "song", name="03 Falling Off the Face of the Earth",
                playbackSeconds="3:36", key="Falling Off the Face of the Earth"},
                {type = "song", name="04 Far From Home", playbackSeconds="3:48", key="Far From
                Home"},
                {type = "song", name="05 It's a Dream", playbackSeconds="6:32", key="It's a
                Dream"},
                {type = "song", name="06 Prairie Wind", playbackSeconds="7:35", key="Prairie
                Wind"},
                {type = "song", name="07 Here for You", playbackSeconds="4:33", key="Here for
                You"},
                {type = "song", name="08 This Old Guitar", playbackSeconds="5:33", key="This Old
                Guitar"},
                {type = "song", name="09 He Was the King", playbackSeconds="6:09", key="He Was the
                King"},
                {type = "song", name="10 When God Made Me", playbackSeconds="4:06", key="When God
                Made Me"},
        }
```

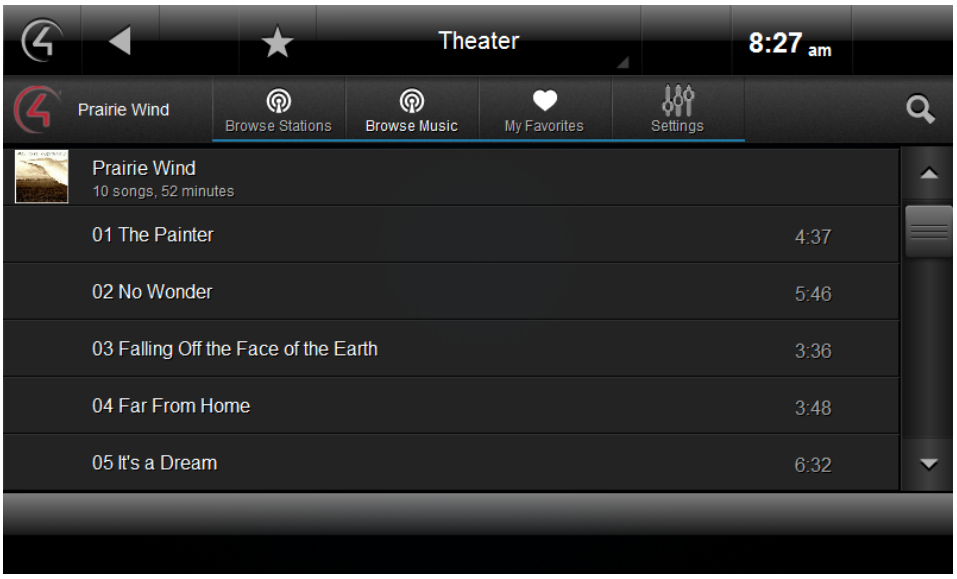
The command then takes this data and uses the helper function BuildListXML to create a list of key value pairs and place then into another variable called "list". Then it creates a final variable called "data" where data is equal to both collection and list. Let's look at the output content for Data and see how collection and list data is delivered. Here is the DATA output from the user selecting Prairie Wind:

```
DATA:  <Collection><type>album</type><name>Prairie Wind</name><key>Prairie Wind</key><artist>Neil
Young</artist><info>10 songs, 52 minutes</info></Collection><List><item><key>The
Painter</key><type>song</type><name>The
Painter</name><playbackSeconds>3:35</playbackSeconds></item><item><key>No
Wonder</key><type>song</type><name>No
Wonder</name><playbackSeconds>3:39</playbackSeconds></item><item><key>Falling Off the Face of the
Earth</key><type>song</type><name>Falling Off the Face of the
Earth</name><playbackSeconds>3:51</playbackSeconds></item><item><key>Far From
Home</key><type>song</type><name>Far From
Home</name><playbackSeconds>3:35</playbackSeconds></item><item><key>It's a
Dream</key><type>song</type><name>It's a
Dream</name><playbackSeconds>3:35</playbackSeconds></item><item><key>Prairie
Wind</key><type>song</type><name>Prairie
Wind</name><playbackSeconds>3:35</playbackSeconds></item><item><key>Here for
You</key><type>song</type><name>Here for
You</name><playbackSeconds>3:35</playbackSeconds></item><item><key>This Old
Guitar</key><type>song</type><name>This Old
Guitar</name><playbackSeconds>3:35</playbackSeconds></item><item><key>He Was the
King</key><type>song</type><name>He Was the
King</name><playbackSeconds>3:35</playbackSeconds></item><item><key>When God Made
Me</key><type>song</type><name>When God Made
Me</name><playbackSeconds>3:35</playbackSeconds></item></List>
```

If we take the <Collection> part of the output and arrange it in XML it would look like this:

```
<Collection>
        <type>album</type>
        <name>Prairie Wind</name>
        <key>Prairie Wind</key>
        <artist>Neil Young</artist>
        <info>10 songs, 52 minutes</info>
</Collection>
```

This is the data that populates the top part of our collection screen:



If we take the <List> part of the output and arrange it in XML it would look like this:

```
<List>
	<item>
		<key>The Painter</key>
		<type>song</type>
		<name>The Painter</name>
		<playbackSeconds>3:35</playbackSeconds>
	</item>

	<item>
		<key>No Wonder</key>
		<type>song</type><name>No Wonder</name>
		<playbackSeconds>3:39</playbackSeconds>
	</item>

	<item>
		<key>Falling Off the Face of the Earth</key>
		<type>song</type><name>Falling Off the Face of the Earth</name>
		<playbackSeconds>3:51</playbackSeconds>
	</item>

	<item>
		<key>Far From Home</key>
		<type>song</type>
		<name>Far From Home</name>
		<playbackSeconds>3:35</playbackSeconds>
	</item>

	<item>
		<key>It's a Dream</key>
		<type>song</type>
		<name>It's a Dream</name>
		<playbackSeconds>3:35</playbackSeconds>
	</item>

	<item>
		<key>Prairie Wind</key>
		<type>song</type>
		<name>Prairie Wind</name>
		<playbackSeconds>3:35</playbackSeconds>
	</item>

	<item>
		<key>Here for You</key>
		<type>song</type>
		<name>Here for You</name>
		<playbackSeconds>3:35</playbackSeconds>
	</item>

	<item>
		<key>This Old Guitar</key>
		<type>song</type>
		<name>This Old Guitar</name>
		<playbackSeconds>3:35</playbackSeconds>
	</item>

	<item>
		<key>He Was the King</key>
		<type>song</type>
		<name>He Was the King</name>
		<playbackSeconds>3:35</playbackSeconds>
		</item>
	<item>
		<key>When God Made Me</key>
		<type>song</type>
		<name>When God Made Me</name>
		<playbackSeconds>3:35</playbackSeconds>
```

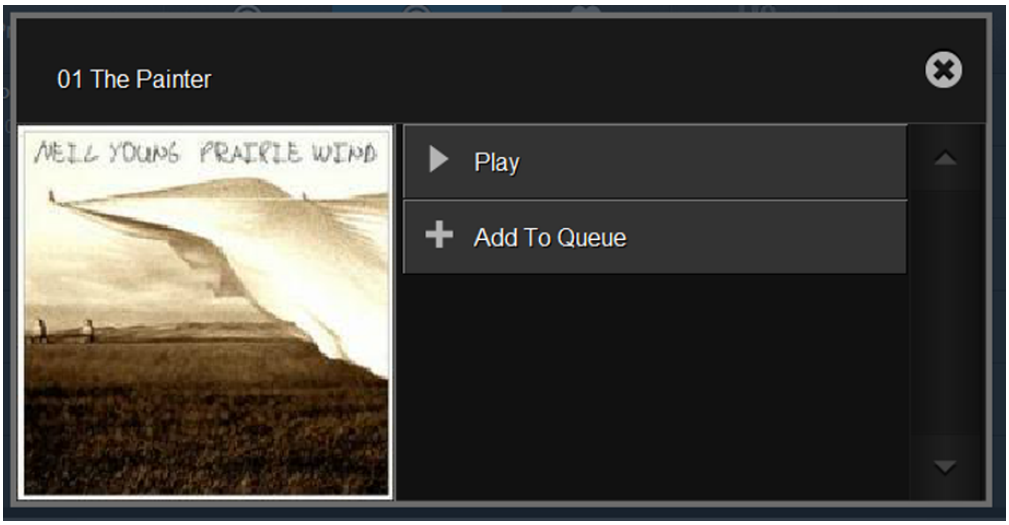
```
        </item>
</List>
```

At the end of our command definition we can see where we use the DATA_RECIEVED helper function within the ReceivedfromProxy command to load the data onto our "Prairie Wind" collection screen. We see the Prairie Wind album, total songs and length followed by all of the songs and their respective length.

Selecting an Item from a Collection Screen (Play and Add to Queue)

The next option presented to our user from the Hello World driver is the selection of a song from the Prairie Wind album's Collection Screen. When a song is selected from the album, the user is provided a choice of playing the song or adding the song to a queue. A selection to play the song will initiate the Now Playing screen and stream the selected audio. Selecting to add the song to the queue will add the song into a queue and return the user to the Prairie Wind album's Collection Screen. Several things happen within our driver before we get to that point including the use of an MSP Action.

Typically, we begin by looking at lua output in ComposerPro to help explain the Hello World driver functionality. However, when the user selects the song "Prairie Wind" from the UI you'll notice that no lua output is displayed. You will notice that the UI changes. It is updated with a screen as seen below:



This screen is the result of an MSP Action being invoked. In the previous section we looked at the first half of the PlayMusicCollection xml to see how a collection screen was populated containing the Prairie Wind songs. That first half of this xml uses a Data Command just like the BrowseMusicCollection screen we previously discussed.

Now, let's look at the second half of that xml to see how an Action is defined and ultimately displayed on the UI. Here is the last section of the xml for PlayMusicCollection. Note the definition of two ActionIds: Play Track & AddTrack to Queue.

To understand how these are used we can use lua debug. When the user selects the Play button on the Action screen our output looks like this:

ReceivedFromProxy(): play on binding 5001; Call Function play()
ROOMID: 521
SEQ: 5
ARGS: <args><arg name="key">The Painter</arg><arg name="playType">PLAY</arg></args>
NAVID: 15
SendToProxy (5001, SEND_EVENT)
NAVID: 15
NAME: QueueChanged
EVTARGS: <NowPlayingIndex>0</NowPlayingIndex><List><item><Id>1</Id><SubTitle>Neil Young - Prairie Wind</SubTitle><Title>01 The Painter</Title><URL>http://192.168.0.129/driver/HelloWorld/media/Neil%20Young/Prairie%20Wind/01%20The%20Painter%20clip.mp3</URL><is_preset>false</is_preset><ImageUrl>http://192.168.0.129/driver/HelloWorld/media/Neil Young/Prairie Wind/Folder.jpg</ImageUrl><key>The Painter</key><type>song</type></item></List>
SendToProxy (5001, SELECT_INTERNET_RADIO)
STATION_URL: http://192.168.0.129/driver/HelloWorld/media/Neil%20Young/Prairie%20Wind/01%20The%20Painter%20clip.mp3

In the first line of the output we can see that we are invoking the play function. There is a defined Action in our driver's xml that is invoked when the Play button is pressed. It looks like this:

```
<Action>
        <Id>PlayTrack</Id>
        <Name>Play</Name>
        <IconId>lis_rha_ico_play.png</IconId>
        <Command>
          <Name>play</Name>
          <Type>PROTOCOL</Type>
          <Params>
            <Param>
              <Name>key</Name>
              <Type>FIRST_SELECTED</Type>
              <Value>key</Value>
            </Param>
            <Param>
              <Name>playType</Name>
              <Type>DEFAULT</Type>
              <Value>PLAY</Value>
            </Param>
            </Params>
        </Command>
      </Action>
<Action>
```

This action is named Play and that's what we see on the Action screen. It also calls a function called "play". That is the same function we see called in the beginning of our lua output. It also sends a playType of "PLAY". We can also see this in our lua output. Finally notice the key value of "The Painter". The combination of these elements not only plays the song but also add the song to the queue. The Hello World driver is designed to not only build a queue based on the user selection of the "Add To Queue" UI button, but also place a song into the queue when it is played.

If we look at our driver's lua code we can see the play function defined as:

```
function PRX_CMD.play(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)

		--clear NowPlaying  if station is playing
		if (gNowPlaying[1] ~= nil) then
			if (gNowPlaying[1].type == "station") then
				gNowPlaying = {}
				gCurrentSongIndex = 1
			end
		end

		--Add Song info to gNowPlaying Queue
		table.insert(gNowPlaying, g_MediaByKey[args.key])
		gNowPlaying[#gNowPlaying].Id = #gNowPlaying

		local NowPlayingIndex = BuildSimpleXml(nil, {["NowPlayingIndex"] = #gNowPlaying -
		1})
		local List = BuildListXml(gNowPlaying, true)
		local NowPlaying = "<NowPlaying><actionIds>Preset</actionIds>" ..
		"<key>" ..     gNowPlaying[gCurrentSongIndex].key .. "</key>" ..
		"<type>" .. gNowPlaying[gCurrentSongIndex].type .. "</type>" ..
		"<is_preset>" .. gNowPlaying[gCurrentSongIndex].is_preset .. "</is_preset>" ..
		"</NowPlaying>"
		local data = NowPlayingIndex .. List .. NowPlaying
		QueueChanged(idBinding, nil, tParams["ROOMID"], data)

	if (args.playType == "PLAY") then
		gCurrentSongIndex = #gNowPlaying
		local id = gCurrentSongIndex

		--***Select the Song to Play ***
		SelectInternetRadio(idBinding, tParams["ROOMID"], gNowPlaying[id].URL, id)

		--***UpdateMediaInfoForRoom***
		UpdateMediaInfo(idBinding, gNowPlaying[id].Title, gNowPlaying[id].SubTitle, "Line
		3", "Line 4", gNowPlaying[id].ImageUrl, tParams["ROOMID"], "secondary", "True")

		--***Goto Now Playing screen***
		local nextscreen = "<NextScreen>#nowplaying</NextScreen>"
		DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)
	elseif (args.playType == "ADD_TO_QUEUE") then
		--no action needed, the user will stay on the BrowseMusicCollection screen
	else
		print("PRX_CMD.play(), unhandled playType: " .. args.playType)
	end

end
```

Note the section of code where the song info is added to gNowPlaying Queue. This is where the song is added into NowPlaying (Queue) tracking table called gNowPlaying.

You can also see where BuildListXml takes gNowPlaying and formats it into xml to be used by the QueueChanged event. This example of QueueChanged includes the optional NowPlaying element.

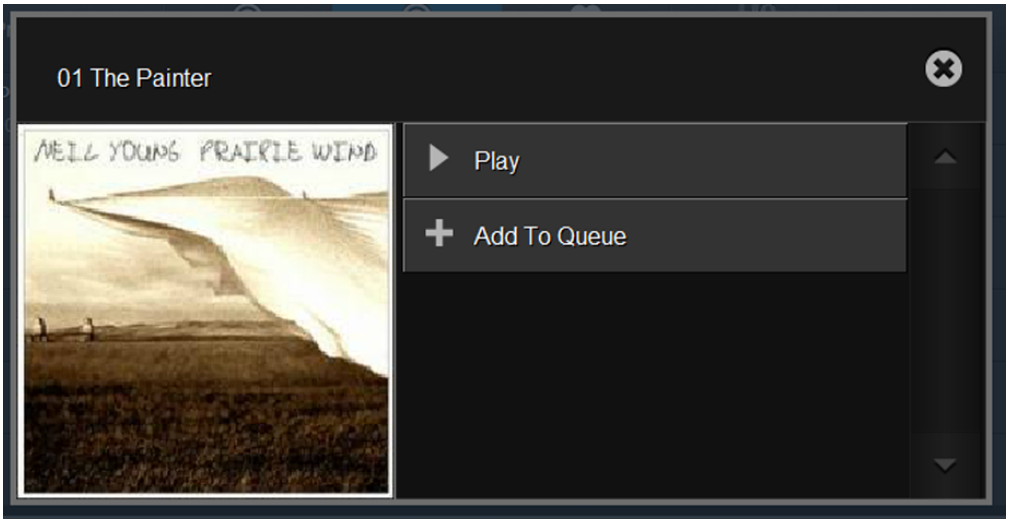
This is followed by the <NowPlaying> element and all of the Action's parameters.
Next you see that the QueueChanged event is sent up to the Proxy to update the UI.

Remember that our Action was defined with a Play Type default parameter of "PLAY"? You can see that the play function has an argument of "PLAY". If it is PLAY the SelectInternetRadio command is fired. This initiates the audio stream for Prairie Wind.

Next we can see the use of a helper function called UpdateMediaInfo. This function populates the room media information with the song data such as title, subtitle and artwork. This is pulled from the gNowPlaying table and sent to the room using the ROOMID parameter.

The function also invokes the nowplaying screen to display the data.

To review, Our Action screen was displayed when a user selected the song Prairie Wind. The user was presented with two options: Play and Add to Queue:



Let's say that the user selected the Add to Queue option. We would see the following lua output:

```
ReceivedFromProxy(): play on binding 5001; Call Function play()
ROOMID:  6
ARGS:  <args><arg name="key">The Painter</arg><arg
name="playType">ADD_TO_QUEUE</arg></args>
SEQ:  15
NAVID:  10
SendToProxy (5001, SEND_EVENT)
NAME:  QueueChanged
EVTARGS:  <NowPlayingIndex>0</NowPlayingIndex><List><item><Id>1</Id><SubTitle>Neil Young -
Prairie Wind</SubTitle><Title>01 The
Painter</Title><URL>http://10.14.14.115/driver/HelloWorld/media/Neil%20Young/Prairie%20Wind
/01%20The%20Painter%20clip.mp3</URL><is_preset>false</is_preset><ImageUrl>http://10.14.14.1
15/driver/HelloWorld/media/Neil Young/Prairie Wind/Folder.jpg</ImageUrl><key>The
Painter</key><type>song</type></item></List>
```

This is very similar to the output displayed when they selected Play. Most notably is the function being called; play. The same function that was called in the previous example.

Here is what the xml looks like from our driver for the Action used when the Add To Queue button is selected:

```
<Action>
      <Id>AddTrackToQueue</Id>
      <Name>Add To Queue</Name>
      <IconId>lis_rha_ico_add.png</IconId>
      <Command>
       <Name>play</Name>
       <Type>PROTOCOL</Type>
       <Params>
        <Param>
         <Name>key</Name>
         <Type>FIRST_SELECTED</Type>
         <Value>key</Value>
        </Param>
        <Param>
         <Name>playType</Name>
         <Type>DEFAULT</Type>
         <Value>ADD_TO_QUEUE</Value>
        </Param>
       </Params>
      </Command>
     </Action>
```

This action is named Add To Queue and that's what we see on the Action screen. Once again, it also calls a function called play. That is the same function we see called in the beginning of our lua output. It also sends a playType of ADD_TO_QUEUE. We can also see this in our lua output. Finally notice the key value of "The Painter". This time The Painter is only added to the queue. It is not played.

Refreshing Screens

Note that the RequiresRefresh XML tag can be included in the screen's XML to refresh the screen by resending the original DataCommand for a normal screen or by sending a GetQueue for the now playing screen. This response is NOT supported on commands executed by notifications.

```
<RequiresRefresh>true</RequiresRefresh>
```

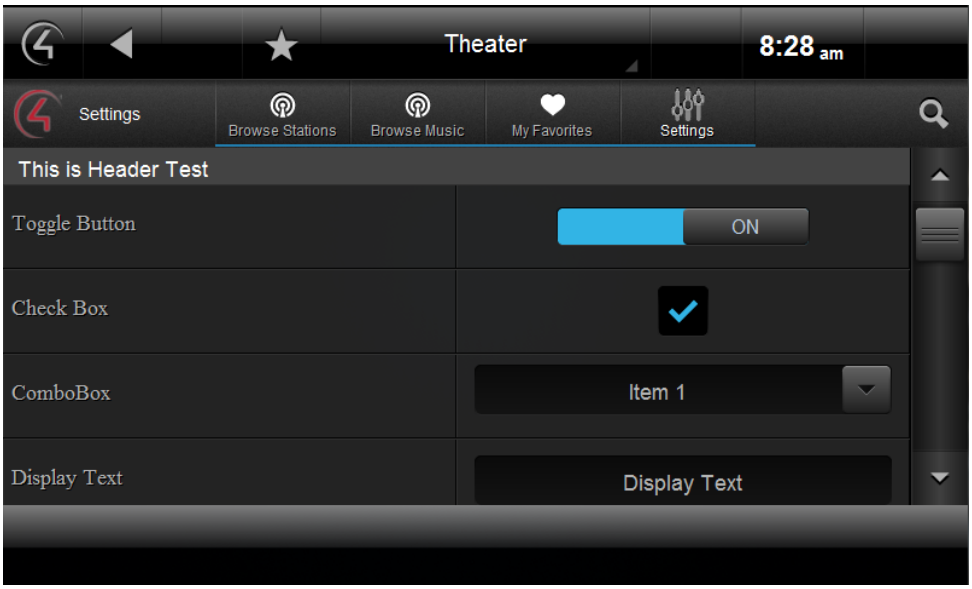
Let's look at our driver's lua code once again to see how the play function is used in this instance:

```
function PRX_CMD.play(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)

		--clear NowPlaying  if station is playing
		if (gNowPlaying[1] ~= nil) then
			if (gNowPlaying[1].type == "station") then
				gNowPlaying = {}
			end
		end

		--Add Song info to gNowPlaying Queue
		table.insert(gNowPlaying, g_MediaByKey[args.key])
		gNowPlaying[#gNowPlaying].Id = #gNowPlaying

		local NowPlayingIndex = BuildSimpleXml(nil, {["NowPlayingIndex"] =
		gCurrentSongIndex - 1})
		local List = BuildListXml(gNowPlaying, true)
		local NowPlaying = "<NowPlaying><actionIds>Preset</actionIds></NowPlaying>"
		local data = NowPlayingIndex .. List --.. NowPlaying
		QueueChanged(idBinding, nil, tParams["ROOMID"], data)

	if (args.playType == "PLAY") then
		gCurrentSongIndex = #gNowPlaying
		local id = gCurrentSongIndex

		--***Select the Song to Play ***
		SelectInternetRadio(idBinding, tParams["ROOMID"], gNowPlaying[id].URL, id)

		--***UpdateMediaInfoForRoom***
		UpdateMediaInfo(idBinding, gNowPlaying[id].Title, gNowPlaying[id].SubTitle, "Line
		3", "Line 4", gNowPlaying[id].ImageUrl, tParams["ROOMID"], "secondary", "True")

		--***Goto Now Playing screen***
		local nextscreen = "<NextScreen>#nowplaying</NextScreen>"
		DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], nextscreen)
	elseif (args.playType == "ADD_TO_QUEUE") then
		--no action needed, the user will stay on the BrowseMusicCollection screen
	else
		print("PRX_CMD.play(), unhandled playType: " .. args.playType)
	end

end
```

In our first example we looked at the function and how it handled a playType equal to PLAY. This time our playType is equal to ADD_TO_QUEUE. As you can see, in the code and on the UI when Add To Queue is selected the user is returned to the Prairie Wind collection screen.

Next Screen and Building a Settings Type Screen

The last type of screen supported through the Media Service Proxy is a Settings Screen. The Settings Screen Type is useful for allowing users to make driver based configurations from within the user interface. The Hello World driver's sample Settings Screen is access from the top of the UI through the selection of a Tab just like the List Screen Type (Browse Stations tab) and the Collection Screen type (through the Browse Music tab). If we select the Settings Tab we'll see the following UI:



As you can see, the Settings Screen Type is made up of numerous UI elements. Before we look at the code behind the display and execution of each of these elements it's important to understand the intended use of each:

The first component we see at the top of the screen is a Header. This is useful for providing text to support the general purpose of this instance of the Settings Screen.

Next, we can see a Toggle Button– The next component is the Toggle Button. A toggle button can be useful when the need to provide the user a choice of selecting two defined action. For example, our driver supports toggling between On and Off.

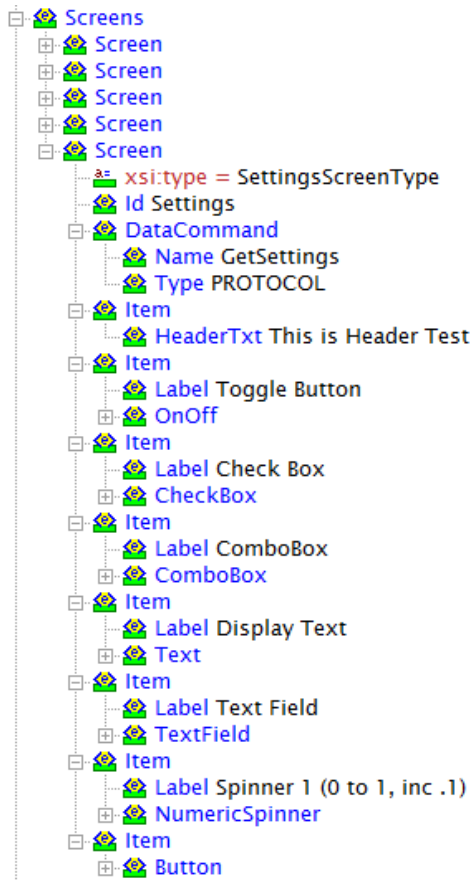
The Toggle Button is followed by a Check Box. The Check Box element supports a populated box which is active (as in the example above) or an unpopulated box which is inactive.

This is followed by a ComboBox. Combo boxes differ slightly based on the UI environment where they are displayed. Generally speaking, Combo Boxes consist of a text field with a drop down which allows the user to select one of the items found within the drop down. Each item inside of the drop down has an ID and a Value. The ID is used to select the list item.

Next, we have a Display Text element. Display Text is text which is displayed on the UI which cannot be edited. This is followed by the Text Field element. Text Fields display text on the UI which can be edited. When a user selects a text field, a keyboard may appear or another UI convention similar to a keyboard will appear to allow them to edit the text displayed within the Text Field.

The Next UI element we see is a Numeric Spinner. Numeric Spinners have two important elements: a Value Range and an Increment Value. The Value Range is an XML list type consisting of two numbers separated by a space. The first number is the start range. The second number is the stop range. This range of numbers sets the window of values that are considered acceptable values. The Increment Value defines the value set when a user presses an Up or Down Arrow. The values entered through the button presses will not fall outside of the window of values defined by the Range.

The last element we see at the bottom of the UI is a Button. This example uses the button to display the setting values the user configures in the Setting Screen. Let's look at how the Setting Screen type is defined in our driver's XML. If we open up the Capabilities ->UI -> Screens -> Settings Screen definition we'll see this:



Here we see all of the UI elements that we have just reviewed. At the top of the definition, we see the Settings Screen Data Command which is executed when the Setting Screen loads. It is named Get Settings. We can see that GetSettings is a command which is sent from the UI, through the MSP and to the Protocol based on its Type. Now, let's load the Setting Screen and look at the lua output in ComposerPro:

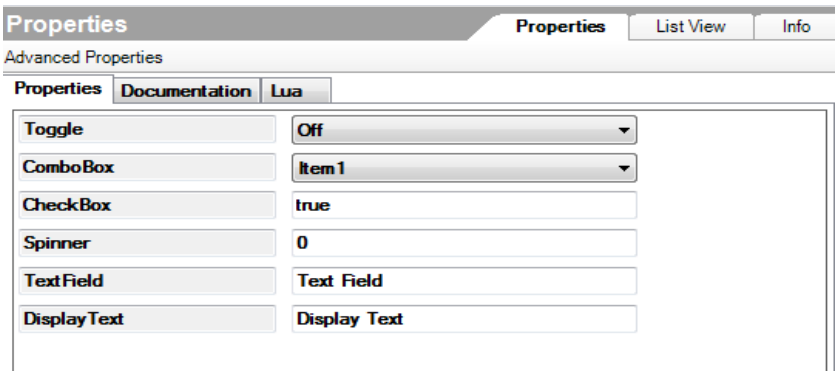
```
ROOMID:  7
ReceivedFromProxy(): GetSettings on binding 5002; Call Function GetSettings()
ARGS:  <args/>
SEQ:  39
NAVID:  11
PRX_CMD.GetSettings(5002, 39) for nav 11
SendToProxy (5002, DATA_RECEIVED)
```

The first line of lua output shows us that the GetSettings Data Command is sent to the protocol using the ReceivedFromProxy API. Let's look at getSetting and see how it's defined:

```
function PRX_CMD.GetSettings(idBinding, tParams)
        print("PRX_CMD.GetSettings(" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
        tParams.NAVID)
        local settings = {}
        for i,v in pairs(Properties) do
                if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
                        or (i == "TextField") or (i == "DisplayText")) then
                        settings[i] = v
                elseif (i == "ComboBox") then
                        --ComboBox value in Navigator is the index of the value in the list
                        --since in our example, the Composer Combo Property values are Item1,
                                Item2, Item3 & Item4
                        --we can just take the last character and typecast it as a number
                        local value = tonumber(string.sub(v,5,5))
                        settings[i] = value
                end
        end
        local data = BuildSimpleXml("Settings", settings, true)
        DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], data)

end
```

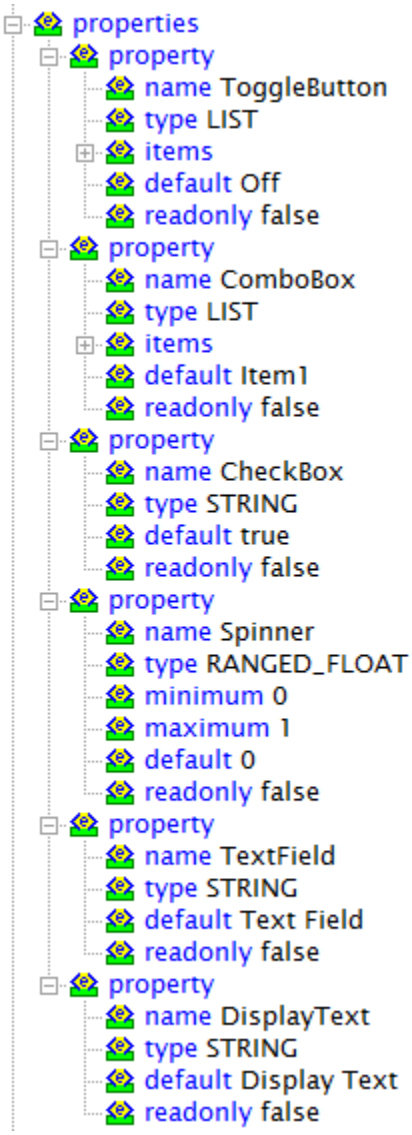
The purpose of GetSettings is to populate the UI elements on the Settings screen with the appropriate data from the protocol. This data is sent to the proxy using the DATA_RECEIVED notification through the SendToProxy API. This data included a populated checkbox, data displayed in the combo box element, any text in the display text area and so on.

Our Hello World driver uses GetSettings (and its supporting code) as a mechanism to store setting data. The settings data is stored in this example in Properties. We can see the properties in the ComposerPro Properties area for the driver:



Storing the data in Properties is useful for several reasons. One is that the values are persisted so the driver does not need to handle that aspect. Also Properties provide a visual presentation of the data in Composer Pro.

Properties are defined in the Hello World driver in the config section of the driver.xml:



The image above is from the hello World driver's xml properties. You can see all of the properties that have been created to handle the data for each of the respective UI elements in the Settings screen. We have named the property the same as the name of the UI element. This will make it easier for us as we review the data behind these properties.

Properties are initialized when a driver loads. They are loaded with all of the default values defined for each property here in the xml. For example, here is how the checkbox property is defined:

```
<property>
  <name>CheckBox</name>
  <type>STRING</type>
  <default>true</default>
  <readonly>false</readonly>
</property>
```

We can see the default value is set to true. This populates the checkbox when the driver loads and that can be seen when the Hello World Settings screen is initially loaded.

Let's look at the GetSettings command again:

```
function PRX_CMD.GetSettings(idBinding, tParams)
	print("PRX_CMD.GetSettings(" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
	tParams.NAVID)
	local settings = {}
	for i,v in pairs(Properties) do
		if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
			or (i == "TextField") or (i == "DisplayText")) then
			settings[i] = v
		elseif (i == "ComboBox") then
			--ComboBox value in Navigator is the index of the value in the list
			--since in our example, the Composer Combo Property values are Item1,
				Item2, Item3 & Item4
			--we can just take the last character and typecast it as a number
			local value = tonumber(string.sub(v,5,5))
			settings[i] = value
		end
	end
	local data = BuildSimpleXml("Settings", settings, true)
	DataReceived(idBinding, tParams["NAVID"], tParams["SEQ"], data)

End
```

Keep in mind, this settings data is sent to the proxy using the DATA_RECEIVED helper function through the SendToProxy API. The data is assembled in a block of xml with the helper function BuildSimpleXml. We can see how this data is formatted in our lua output window:

```
ROOMID:  57
ARGS:  <args/>
SEQ:  43
NAVID:  31468554-9C97-202F-3034-710F3C8DD6C7
PRX_CMD.GetSettings(5001, 43) for nav 31468554-9C97-202F-3034-710F3C8DD6C7
SendToProxy (5001, DATA_RECEIVED)
SEQ:  43
NAVID:  31468554-9C97-202F-3034-710F3C8DD6C7
ROOMID:  57
DATA:  <Settings><TextField>Text
Field</TextField><CheckBox>true</CheckBox><Toggle>Off</Toggle><DisplayText>Display
Text</DisplayText><ComboBox>1</ComboBox><Spinner>0.6</Spinner></Settings>
ARGS:  table: 0x861fff0
ReceivedFromProxy(): GetSettings on binding 5001; Call Function GetSettings()
```

The data passed is formatted as:

```
<Settings>
	<TextField>TextField</TextField>
	<CheckBox>true</CheckBox>
	<Toggle>Off</Toggle>
	<DisplayText>DisplayText</DisplayText>
	<ComboBox>1</ComboBox>
	<Spinner>0.6</Spinner>
</Settings>
```

Note that all of the data shown here is default property data. This data is placed into a data parameter in: `tParams["DATA"] = data` Which is sent to the proxy in DATA_RECEIVED. When the proxy receives it, it then populates the Settings Screen UI.

It's important to understand the components in each of these entries. Each has a Value and a Name. For example, text field entry looks like this: `<TextField>TextField</TextField>`

We can see the name as TextField and the default value as Text Field. This is the data that is assembled as: `<TextField>TextField</TextField>`.

Now let's look at how the data is assembled in in the GetSettings function before it is sent to the BuildSimpleXml helper function. This section of the GetSettings code accomplishes that:

```
local settings = {}
	for i,v in pairs(Properties) do
		if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
			or (i == "TextField") or (i == "DisplayText")) then
			settings[i] = v
		elseif (i == "ComboBox") then
			--ComboBox value in Navigator is the index of the value in the list
			--since in our example, the Composer Combo Property values are Item1,
				Item2, Item3 & Item4
			--we can just take the last character and typecast it as a number
			local value = tonumber(string.sub(v,5,5))
			settings[i] = value
		end
	end
```

The function creates a local table called settings. This is populated from the Properties data in pairs of i & v. To understand this, let's look at how an individual property is defined; in comparison with how the UI element is defined in the Setting Screen xml we'll see that their values are the same. For example, let's look at the Text Field Property again. Here it is defined in the driver's xml:

```
<property>
		<name>TextField</name>
		<type>STRING</type>
		<default>Text Field</default>
		<readonly>false</readonly>
	</property>
<property>
```

Here it is defined under the Setting Screen XML:

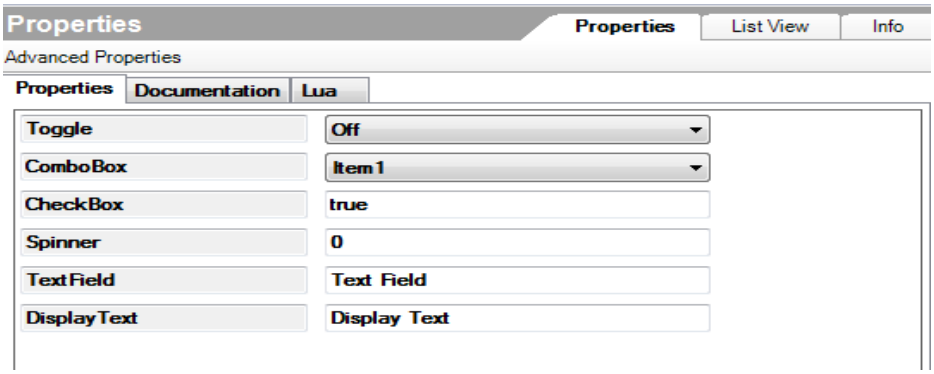
```
<Item>
	<Label>Text Field</Label>
```

```
        <TextField propertyName="TextField"/>
</Item>
```

These being named identically allows our driver to use a loop to build the data for our setting table.

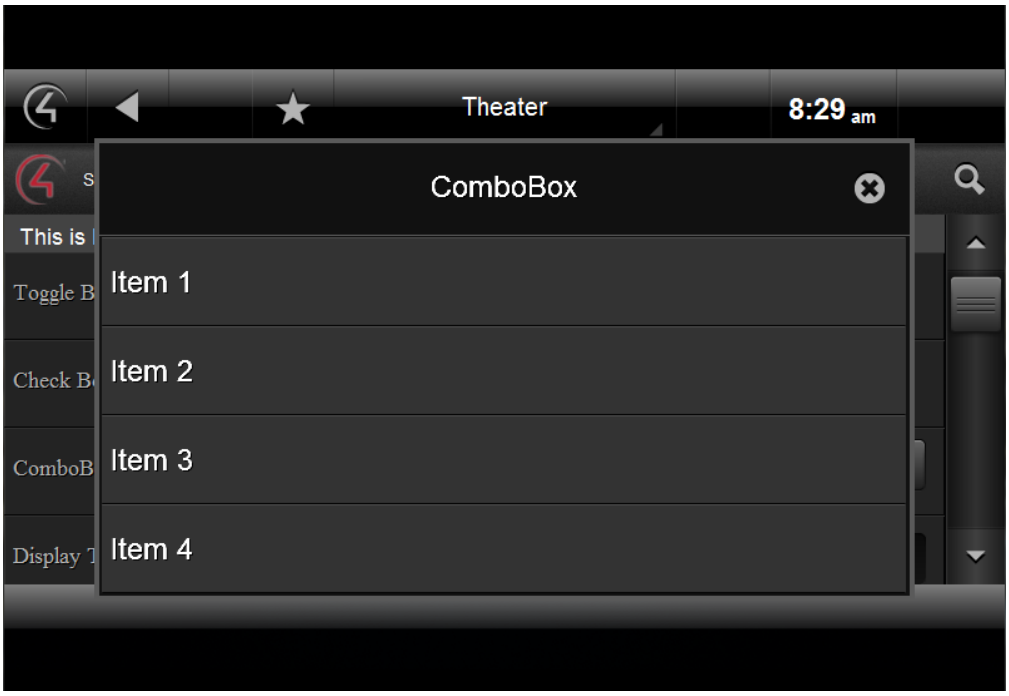
```
local settings = {}
        for i,v in pairs(Properties) do
                if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
                        or (i == "TextField") or (i == "DisplayText")) then
                        settings[i] = v
                elseif (i == "ComboBox") then
                        --ComboBox value in Navigator is the index of the value in the list
                        --since in our example, the Composer Combo Property values are Item1,
                                Item2, Item3 & Item4
                        --we can just take the last character and typecast it as a number
                        local value = tonumber(string.sub(v,5,5))
                        settings[i] = value
                end
        end
```

In the code above, the driver uses a DriverWorks convention named (Properties) which goes through all of the driver's defined Properties and assembled them into a table. As the table is iterated, the driver looks to see if the value is "Toggle", "CheckBox", "Spinner", "TextField" or "DisplayText". If those values are found they are made equal to i or index which is the Property Name. These are names we see on the left hand side of the ComposerPro Properties screen:



We then make the settings table at the property name equal to value: `settings[i] = v` or the value the property currently has. This is how the screen above is populated. To summarize, the driver builds a table called settings. This table consists of indexed entries (i) with the current value that is in the property (v).

This methodology is consistent for all of the Settings Properties except for Combo Box. Combo Box is unique in that when it is selected it displays another screen:
This methodology is consistent for all of the Settings Properties except for Combo Box. Combo Box is unique in that when it is selected it displays an indexed screen of items:



If we look at how Combo Box is defined as a Property we'd see this:

```
<property>
             <name>ComboBox</name>
             <type>LIST</type>
             <items>
               <item>Item1</item>
               <item>Item2</item>
              <item>Item3</item>
              <item>Item4</item>
             </items>
             <default>Item1</default>
             <readonly>false</readonly>
           </property>
           <property>
```

Each of the entries displayed above: Item 1, Item 2, etc. are indexed. For example, Item 1 is indexed as [1], Item 2 is indexed as [2] and so on. When a user selects on the corresponding index value is sent. The GetSetting function has code to remove the "Item" portionof each of the elements and handle just its indexed value. So, Item 2 becomes 2. These are the values that need to be passed into our setting stable.

This is accomplished with this code:

```
local value = tonumber(string.sub(v,5,5))
```

The code above uses the lua convention called string.sub for the string value (v) of the item. String.sub removes everything but the data in the fifth spot – which is just the number or the item's index value. So, Item2 after string.sub is just 2. Then the lua function tonumber takes that string value of 2 and turns it into the numerical version of 2 and places it in the table.

Now that all of the Setting Properties and their values have been assembled in a table, GetSettings uses the BuildSimpleXml to create xml with the start and end tags of

<Settings></Settings> It all gets wrapped in the "DATA" table and passed into tParams which is sent to the UI using the DATA_RECEIVED helper function through the SendToProxy API.

```
local data = BuildSimpleXml("Settings", settings, true)
	tParams["DATA"] = data
	SendToProxy(idBinding, "DATA_RECEIVED", tParams)
End
```

At this point we've provided a Setting Screen UI for the User to interact with. Now, let's examine what happens when a Setting Property is changed buy the user. Let's say that the check box property is changed on our UI. If we un-populate the check box from the setting screen and look at our lua output we'll see this:

```
ReceivedFromProxy(): SettingChanged on binding 5001; Call Function SettingChanged()
ROOMID:  7
ARGS:  <args><arg name="PropertyName">CheckBox</arg><arg name="Value">false</arg><arg
name="ScreenId">Settings</arg></args>
SEQ:  8
NAVID:  11
PRX_CMD.SettingChanged (5001, 8) for nav 11
```

We can see that SettingChanged function is sent through the ReceivedFromProxy API down to the protocol. Along with that are a set of arguments (ARGS). You can see that the ARGS contains a PropertyName a Value and a Screen Id.

Our Hello World driver has a function to handle the SettingChanged command. It looks like this:

```
function PRX_CMD.SettingChanged(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)
	print("PRX_CMD.SettingChanged (" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
tParams.NAVID)
	local propertyName = args["PropertyName"]
	local propertyValue = args["Value"]
	local i = propertyName
	if (Properties[propertyName] ~= nil) then
		if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
			or (i == "TextField") or (i == "DisplayText")) then
				C4:UpdateProperty(propertyName, propertyValue)
		elseif (i == "ComboBox") then
				C4:UpdateProperty(propertyName, "Item" .. propertyValue)
		end
	end
end
```

The first thing the command does is parse the arguments (ARGS) it receives. It then creates two local properties for the Value and the Property Name. The Property Name that is parses is the property name defined in the screen xml. For example, the property name for the checkbox can be seen below:

```
<Item>
	<Label>Check Box</Label>
	<CheckBox propertyName="CheckBox"/>
</Item>
```

The value data is the setting that the property currently has as a result of the user interaction. Our checkbox property can have a setting of true or false. Un-populating the checkbox passes a value of "false". "false" is the value sent down in the ARGS section of the

lua output. Finally, the Screen Id value is the ID value for the screen itself. If we look at the xml definition of the Settings screen we see an ID of "Settings":

```
<Screen xsi:type="SettingsScreenType">
      <Id>Settings</Id>
      <DataCommand>
            <Name>GetSettings</Name>
            <Type>PROTOCOL</Type>
      </DataCommand>
```

That value is important as it dictates which screen the settings have changed in. If there were several settings screens used in our driver, the ID it would be important so the driver would know which settings screen has been changed.

Now that the argument is extracted we have a PropertyName and a Value defined. In our example the Property name is Checkbox and the value is false.

The command then has an argument to make sure that the property is valid:

```
function PRX_CMD.SettingChanged(idBinding, tParams)
        local args = ParseProxyCommandArgs(tParams)
        print("PRX_CMD.SettingChanged (" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
tParams.NAVID)
        local propertyName = args["PropertyName"]
        local propertyValue = args["Value"]
        local i = propertyName
        if (Properties[propertyName] ~= nil) then
                if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
                        or (i == "TextField") or (i == "DisplayText")) then
                                C4:UpdateProperty(propertyName, propertyValue)
                elseif (i == "ComboBox") then
                                C4:UpdateProperty(propertyName, "Item" .. propertyValue)
                end
        end
end
```

The line highlighted above verifies that the property name is something other than nil. If it is (and in this example it is Checkbox) the command calls the C4:UpdateProperty API. This API takes the Name and Value and updates the UI.

The exception to this is the ComboBox property. If we select an item from the ComboBox, Item 3 for example, and look at the lua output we'll see this:

```
ReceivedFromProxy(): SettingChanged on binding 5001; Call Function SettingChanged()
ROOMID:  7
ARGS:  <args><arg name="PropertyName">ComboBox</arg><arg name="Value">3</arg><arg
name="ScreenId">Settings</arg></args>
NAVID:  46C5EA59-BE7F-9EBA-3DA7-75C9D4E050A7
SEQ:  2
PRX_CMD.SettingChanged (5001, 2) for nav 46C5EA59-BE7F-9EBA-3DA7-75C9D4E050A7
```

We can see that once again the SettingsChanged function is called. The arguments contain the Property name of ComboBox and a Value of 3. This is another example of the Item string being converted into an indexed number – 3 in our case. Let's look at the SettingsChanged code again:

```
function PRX_CMD.SettingChanged(idBinding, tParams)
	local args = ParseProxyCommandArgs(tParams)
	print("PRX_CMD.SettingChanged (" .. idBinding .. ", " .. tParams.SEQ .. ") for nav " ..
tParams.NAVID)
	local propertyName = args["PropertyName"]
	local propertyValue = args["Value"]
	local i = propertyName
	if (Properties[propertyName] ~= nil) then
		if ((i == "ToggleButton") or (i == "CheckBox") or (i == "Spinner")
			or (i == "TextField") or (i == "DisplayText")) then
				C4:UpdateProperty(propertyName, propertyValue)
		elseif (i == "ComboBox") then
				C4:UpdateProperty(propertyName, "Item" .. propertyValue)
		end
	end
end
```

The code highlighted above shows us how the command handles a change to a ComboBox. If the Property name is not nil and equal to "ComboBox", then the code updates the property value for the property name and adds an "Item" string in front of the value. Remember that our value is the indexed value of the item selected, which in our example is 3. This updates the data sent to the UI as "Item3"

This can be verified by selected Item 3 from the UI and verifying that the ComboBox value in the ComposerPro Properties page changes from Item1 (default) to Item3.

When a response is sent to the UI and the response has <Settings> then those properties it contains will be updated with the new values. If no new values need to be returned, an empty response is fine. In general every command received from the driver should get a corresponding response.

The last Settings element we need to discuss in the Settings Screen is the Submit button. Buttons are simply another type of item defined in our xml. Here is how the Submit button is defined in our Settings Screen xml:

```
<Item>
	Button>
	<Name>Submit</Name>
	<Command>
		<Name>SubmitButtonCommand</Name>
		<Type>PROTOCOL</Type>
	</Command>
</Button>
</Item>
```

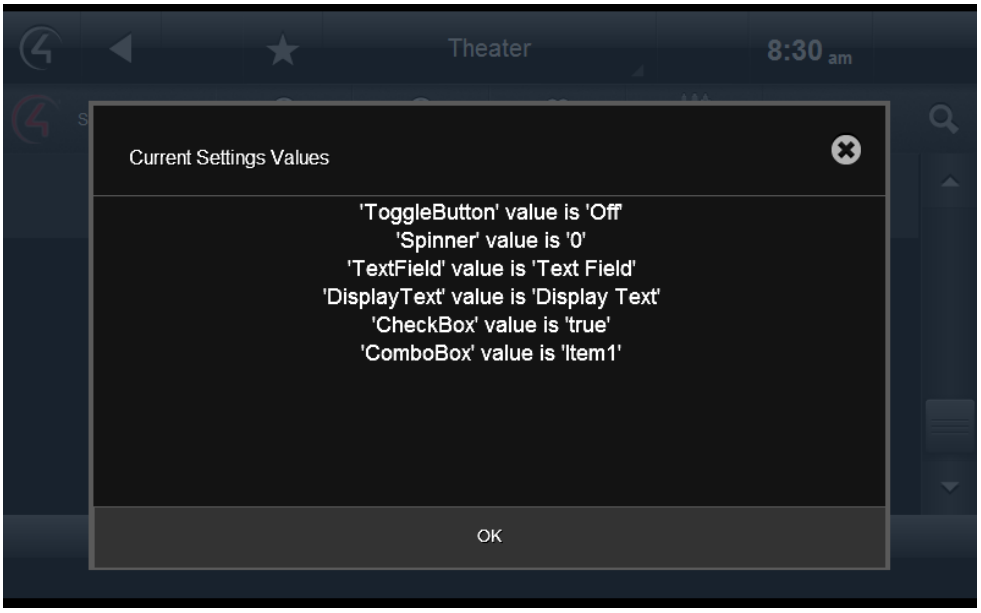
As you can see, Buttons have a name and can have a command associated with them. The command for our Submit button is called SubmitButtonCommand and it is a protocol command meaning that it sent from the proxy to the protocol. When the Submit button is selected from the UI, the lua output shown looks like this:

```
ReceivedFromProxy(): SubmitButtonCommand on binding 5001; Call Function SubmitButtonCommand()
ROOMID:  7
ARGS:  <args/>
SEQ:  3
NAVID:  46C5EA59-BE7F-9EBA-3DA7-75C9D4E050A7
SendToProxy (5001, SEND_EVENT)
```

We can see where the SubmitButtonCommand is called and sent to the protocol on the RecievedFromProxy API. Note that it contains no arguments or ARGS data. Let's see how the SubmitButtonCommand is handled in our Hello World driver:

```
function PRX_CMD.SubmitButtonCommand(idBinding, tParams)
        local out_table = {}
        for k,v in pairs(Properties) do
                table.insert(out_table, "'" .. k .. "' value is '"  .. v .. "'")
        end
        local message = table.concat(out_table, "\n")
        local tArgs = {}
        tArgs["Id"] = "SettingsNotification"
        tArgs["Title"] = "Current Settings Values"
        tArgs["Message"] = message
        tParams["EVTARGS"] = BuildSimpleXml(nil, tArgs, true)
        tParams["NAME"] = "DriverNotification"
        SendToProxy(idBinding, "SEND_EVENT", tParams, "COMMAND")

end
```

The command sends its data to the proxy using the SEND_EVENT notification through the SendToProxy API. It also fires a DriverNotification which displays the summary screen:



A DriverNotification is another element defined in the xml of the MSP. The notification to display the screen above is defined as:

```
DriverNotifications>
        <Notification>
        <Id>SettingsNotification</Id>
                <IconId>PLAY</IconId>
                <Buttons>
                        <Button>
                                <Name>OK</Name>
                                <ScreenId>Settings</ScreenId>
                        </Button>
                </Buttons>
        </Notification>
```

This notification has one button named OK. When that is pressed we stay on the Settings screen. That is defined by the screen name in the ID element.

We can see in the command code that the Settings Notification: DriverNotification is called:

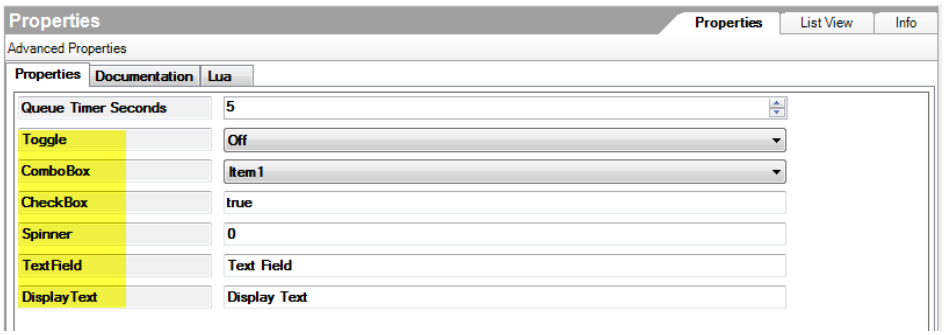
```
function PRX_CMD.SubmitButtonCommand(idBinding, tParams)
        local out_table = {}
        for k,v in pairs(Properties) do
                table.insert(out_table, "'" .. k .. "' value is '"  .. v .. "'")
        end
        local message = table.concat(out_table, "\n")
        local tArgs = {}
        tArgs["Id"] = "SettingsNotification"
        tArgs["Title"] = "Current Settings Values"
        tArgs["Message"] = message
        tParams["EVTARGS"] = BuildSimpleXml(nil, tArgs, true)
        tParams["NAME"] = "DriverNotification"
        SendToProxy(idBinding, "SEND_EVENT", tParams, "COMMAND")

end
```

We also have a title and a message to display. As before, all of this data is placed in tArgs and we use the BuildSimpleXml function to assemble the data.

This section of the code iterates through a local properties table called out_table and extracts all of the current settings values.

```
for k,v in pairs(Properties) do
                table.insert(out_table, "'" .. k .. "' value is '"  .. v .. "'")
        end
```

The code looks for any property that isn't the Queue Timer Seconds property, which is all of the remaining properties in our Properties screen in Composer Pro:

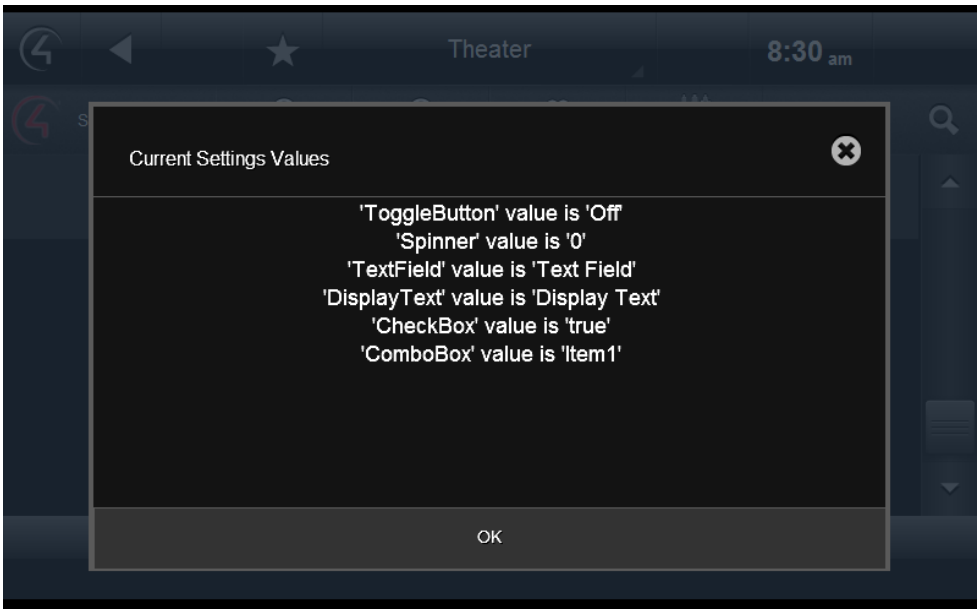


The code inserts a record into this table where k is the name of the property and v is the value. The string "value is" is placed between the name and the value to display 'Toggle' value is 'Off' or 'CheckBox' value is 'false'.

The code then uses the lua element of table.concat to place all of the data in the table into one long string. Then a carriage return is placed after each pair:

```
local message = table.concat(out_table, "\n")
```

This gets placed into a local tArgs table along with the Id, Title and the DriverNotification data and it all gets sent to the proxy through the SEND_EVENT notification through the SendToProxy API. That builds our screen:



Again, if the user selects the OK button, the UI returns to the Settings Screen.

Refreshing Screens

The RequiresRefresh XML tag can be included in the screen's XML to refresh the screen by resending the original DataCommand for a normal screen or by sending a GetQueue for the now playing screen. This response is NOT supported on commands executed by notifications.

```
<RequiresRefresh>true</RequiresRefresh>
```

Dynamic Screen Definitions delivered in 2.9.0

Beginning with OS 2.9.0 the Settings Screen has been enhanced to display, hide or change functionality depending on its data input values. This is supported by allowing dynamic screen definitions in action NextScreen responses. The responses can be formatted as:

```
<NextScreen>
      screenId |
            #defaultScreenId |
            <Screen... (screen xml definition)
</NextScreen>
```

Note: The screen XML definition in the example above is the definition of the screen as it would normally appear in the UI capability and is defined in the MediaServiceProxyUI.xsd schema.

<ReplaceScreen> response
A new <ReplaceScreen> response has been created and delivered in 2.9.0 that will pop the current screen off of the stack and replace it with the defined screen. The ReplaceScreen can be formatted as:

```
<ReplaceScreen>
      screenId |
      Screen...(screen XML definition)
</ReplaceScreen>
```

<RemoveScreen> Response
A screen's data can be made invalid via destructive actions such as a rename or delete. To resolve this, a <RemoveScreen> response has been added in OS 2.9.0 that removes the current screen and goes back to the previous screen. If the screen is a root screen, it calls the current tab's ScreenCommand or loads the current tab's ScreenId.

<RemoveScreen> is also useful if a screen is used as a selection menu. For example, when the selection action is complete and it is desirable to return to the previous screen.

It is recommended to send the <RemoveScreen> response in the follwoing format:

```
"<RemoveScreen>true</RemoveScreen>"
```

Sending this using a shorter syntax can reslut in some Navigators evaluating it to a false or null value.

The DetailType Screen
Information regarding the DetailType Screen has been added to the 3.1.0 version of the documentation. The screen provides the ability to present further details regarding media when the user selects a "View Details" menu option.

For example, a common use case is an album is displayed on the UI. Included in the list of Actions such as Play Now, Add to Queue etc., is an Action called "View Details". When the user selects View Details a NextScreen response is sent which behaves like any other screen change response and the DataCommand (in this case GetDetails) is sent to the Proxy with the specified parameters. This is responded to in the same manner as a regular ListScreen however it includes a Details element as the root instead of a List element.

Below is the DetailType Screen XML definition which needs to added to the driver.xml file:

```
<Screen xsi:type="DetailScreenType">
	<Id>Details</Id>
	<DataCommand>
		<Name>GetDetails</Name>
		<Type>PROTOCOL</Type>
		<Params>
			<Param>
				<Name>screenId</Name>
				<Type>SYSTEM</Type>
				<Value>screenId</Value>
			</Param>
			<Param>
				<Name>tabId</Name>
				<Type>SYSTEM</Type>
				<Value>tabId</Value>
					</Param>
			<Param>
				<Name>id</Name>
				<Type>FIRST_SELECTED</Type>
				<Value>id</Value>
			</Param>
			<Param>
				<Name>itemType</Name>
				<Type>FIRST_SELECTED</Type>
				<Value>itemType</Value>
			</Param>
		</Params>
	</DataCommand>
	<DefaultActionProperty>default_action</DefaultActionProperty>
	<TitleProperty>title</TitleProperty>
	<SubTitleProperty>subtitle</SubTitleProperty>
	<ImageProperty>image</ImageProperty>
	<AttributionImage>attribution_image</AttributionImage>
	<YearProperty>release_date</YearProperty>
	<RatingProperty>rating</RatingProperty>
	<LengthProperty>duration</LengthProperty>
	<Paragraph>
		<HeaderTxt>Description</HeaderTxt>
		<ContentProperty>description</ContentProperty>
	</Paragraph>
</Screen>
```

The following is an example DataRecieved Call for sending Details to the UI:

```
C4:SendToProxy (5001, 'DATA_RECEIVED', {
  SEQ = 171,
  NAVID = '6b782d0ed8d3cc5f-bc919c06-4120-993595660b8f9637030118adac6a',
  DATA = [[
   <Details>
    <description>Norah Jones' debut on Blue Note ...</description>
    <duration>45:06</duration>
    <image>https://static.qobuz.com/images/covers/15/30/0060253753015_600.jpg</image>
    <rating>24 bits / 192.0 kHz - Stereo</rating>
    <release_date>23 septembre 2013</release_date>
    <title>Come Away With Me (24Bit /192 kHz)</title>
    <subtitle>Norah Jones</subtitle>
    <attribution_image>controller://driver/qobuz/icons/device/exper_512.png</attribution_image>
   </Details>
  ]],
}
```

Comparing Device Drivers with Streaming Service Drivers

The Hello World driver used in the previous chapters was created as a driver to demonstrate the capabilities of the Media Service Proxy through a driver created for a media streaming service. Drivers created for device using the MSP are somewhat different. To best understand these differences we recommend loading the following two drivers into your Control4 development environment: HW Selected Device and HW Streaming Device

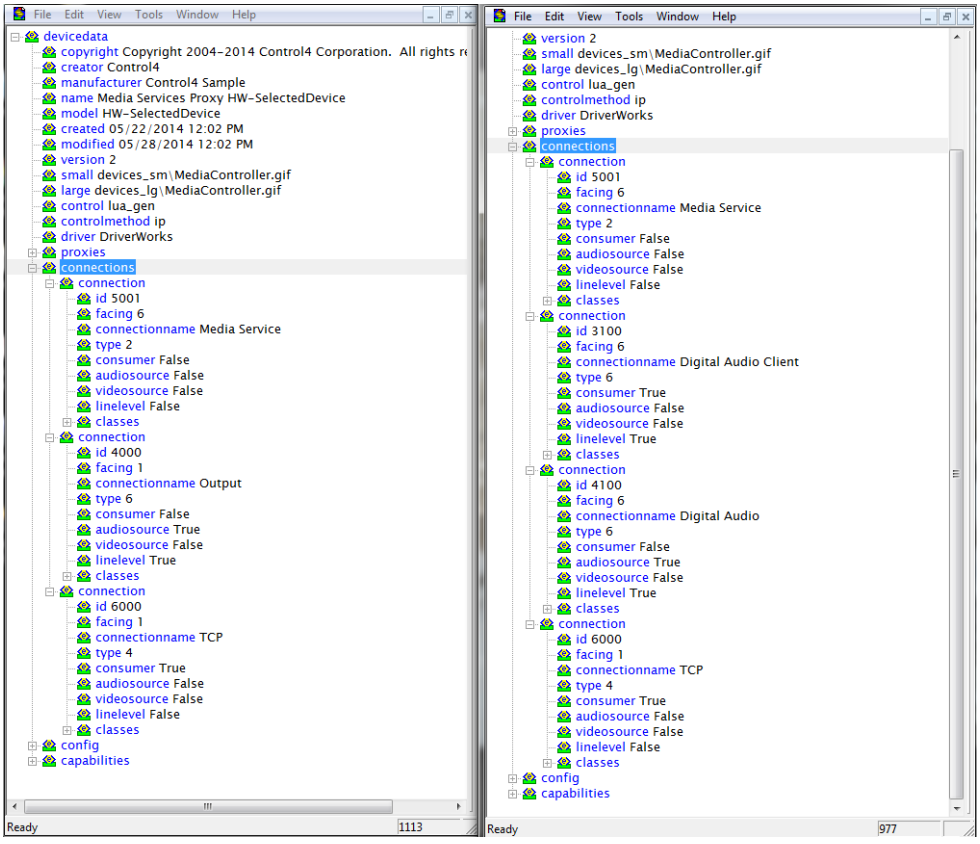
These two drivers are discussed in detail in this chapter. Many of the concepts discussed in this section are detailed in the Hello World driver chapters. Understanding of that content is assumed. If you have not reviewed the section titled: *Understanding Screens and the Media Service Proxy,* we strongly encourage you to do so before continuing.

The best way to understand the important differences between creating a MSP driver for a device versus a streaming service is to focus on the XML and Lua code that make each unique. First, let's look at the Connections XML for each type of driver.

Connections

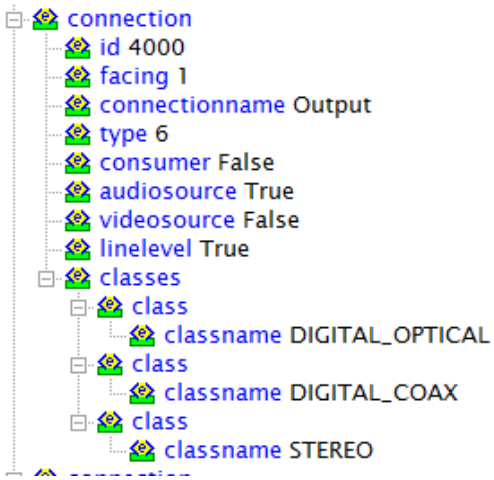
If we compare the Connection XML for both types of drivers we'll see some similarities and some differences:

Device Driver **Streaming Service Driver**

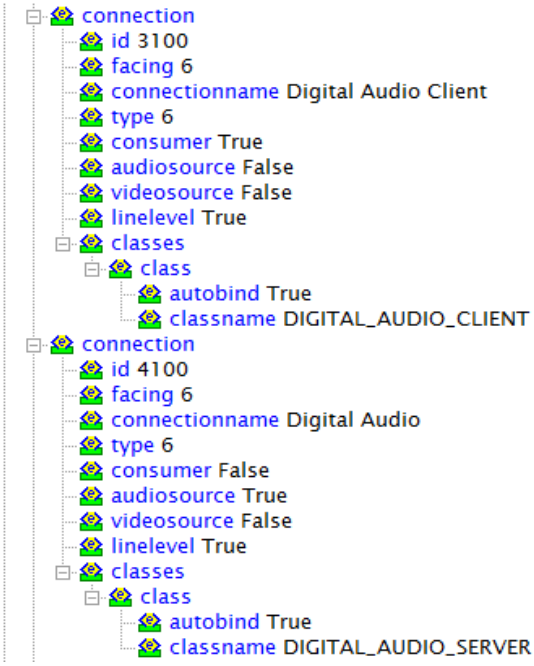


The similarities between the two include a common connection id of 5001. This is the connection for the Media Service proxy and is needed in all drivers. You'll also notice a network TCP connection shared by both drivers.

If you look at Device driver XML on the left you'll see a connection id of 4000. If we expand that connection further we'll see this:

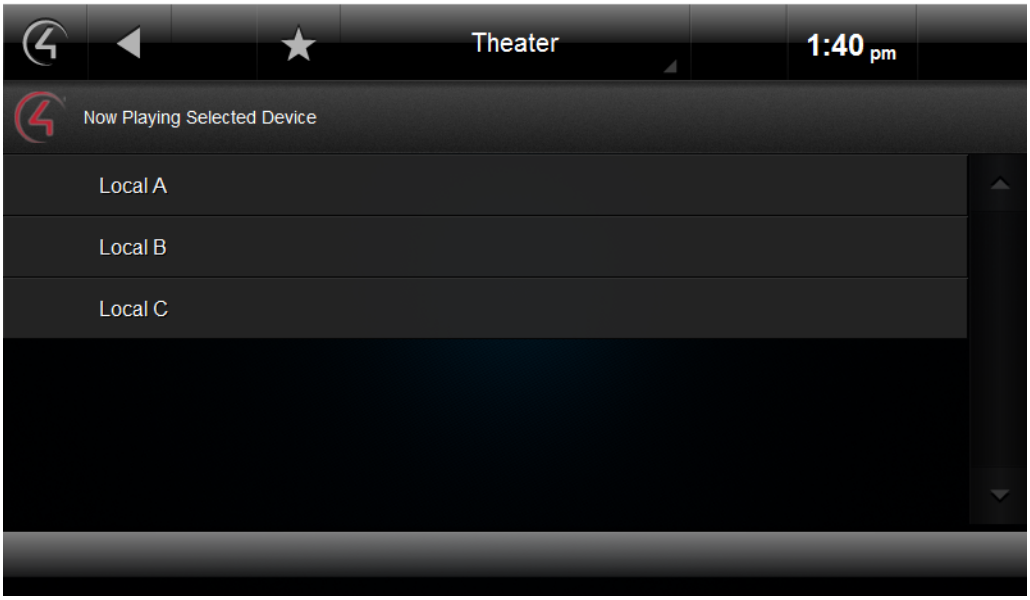


Remember this is an audio device. The sample HW Selected Device driver has one output with three connections classes: DIGITAL_OPTICAL, DIGITAL_COAX and STEREO. Since we consider this device an audio source, it will need to be bound accordingly in a project. This output supports that. Now let's look at the unique connections from the HW Streaming Service driver. Keep in mind that this is a driver for a streaming media service. It doesn't provide audio or video directly through a connection. If we expand the unique connection xml for this driver we'll see this:

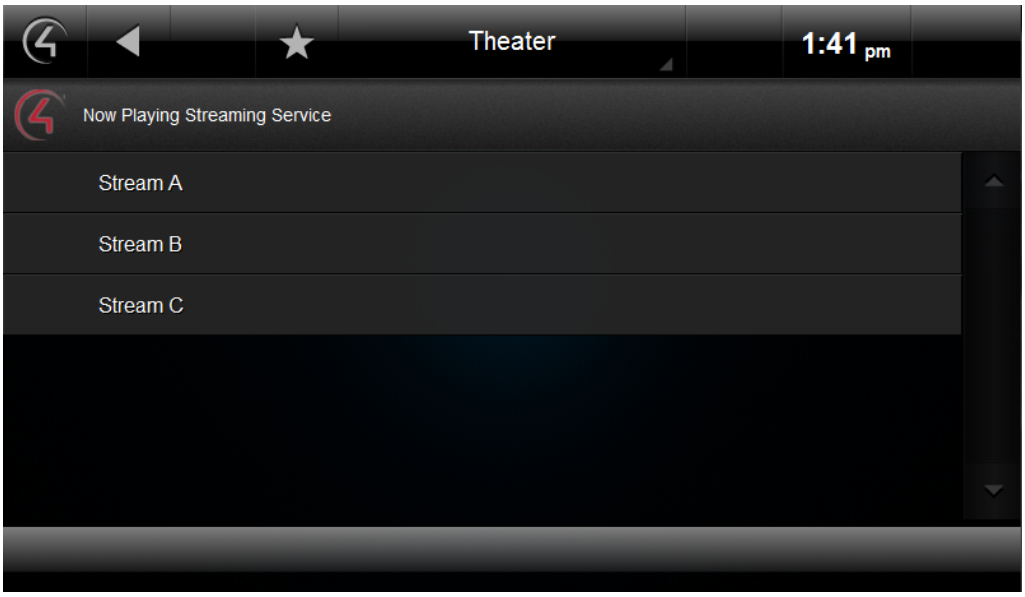


Connection id 3100 is for a Digital Audio Client connection. It is an input for a DIGITAL_AUDIO_CLIENT class. We also see a Digital Audio Connection for Connection id 4100. It is for a second output for a DIGITAL_AUDIO_SERVER class. It is not a hard wired connection since digital media can, in fact, stream to multiple zones. However, that is done at the Director level not the driver level The HW Streaming Service driver is for a server that serves up digital audio to a client. Note the auto bind values set to true for both of these connections. This allows the Director to make the appropriate A/V path connections in ComposerPro when the driver is loaded.

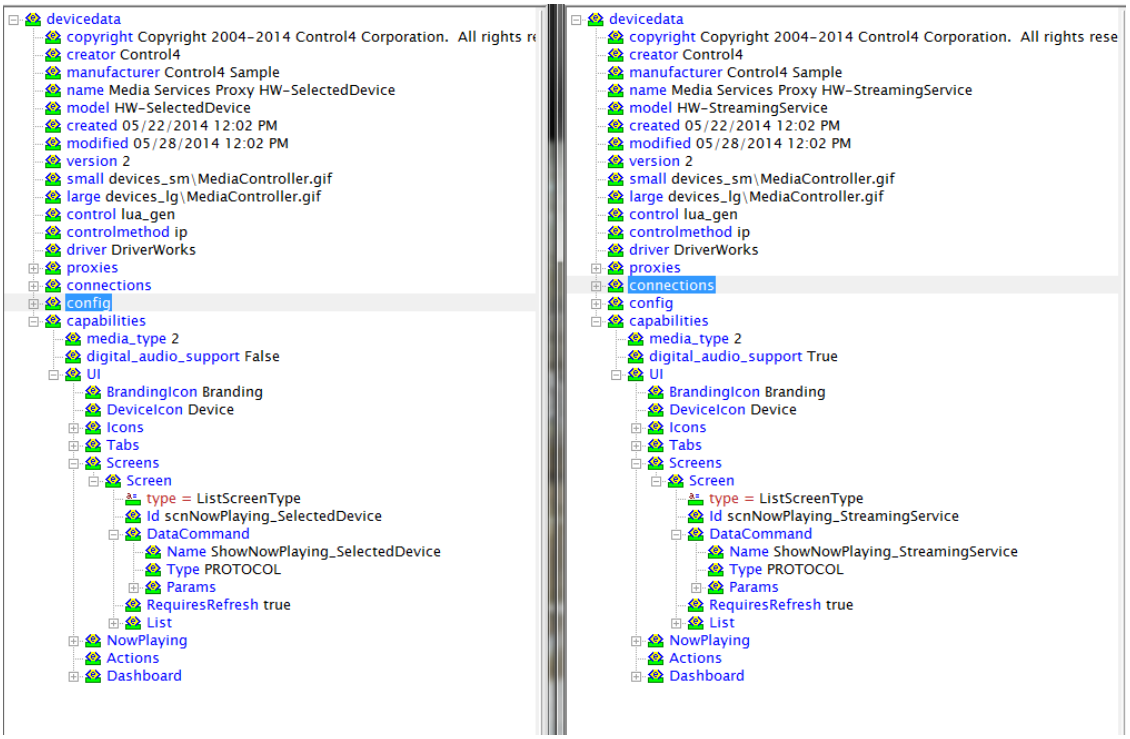
Now is a good time to look at the screens that are loaded for each of the drivers. Remember – Data Commands defined in the XML are fired when a screen loads. Here is the screen for the HW Selected Device driver:



Here is the screen loaded with the HW Streaming Service driver:



They are very similar but loaded with different commands:



Lua

Note the difference in Data Commands for each. If we look at the Lua code for firing the XML for the device driver and the streaming driver we'll find this:

```
function ReceivedFromProxy(idBinding, strCommand, tParams)
        print("--On ReceivedFromProxy: Binding ID = " .. idBinding .. ", Cmd = " .. strCommand)
        if (tParams ~= nil) then PrintTable(tParams) end

        if (idBinding == G_nUI_PROXY_ID) then
```

```
local tResponse = {}
if (tParams ~= nil) then
        tResponse["NAVID"]  = tParams["NAVID"]  -- Required by MSProxy
        tResponse["SEQ"]    = tParams["SEQ"]    -- Required by MSProxy
        tResponse["ROOMID"] = tParams["ROOMID"]
end

if (strCommand == "DESTROY_NAV") then

elseif (strCommand == "ShowNowPlaying_StreamingService") then
        tResponse["DATA"] = "<List><item><name>Stream
        A</name><id>A</id></item><item><name>Stream
        B</name><id>B</id></item><item><name>Stream
        C</name><id>C</id></item></List>"
        SendToProxy(idBinding, "DATA_RECEIVED", tResponse)

elseif (strCommand == "ShowNowPlaying_SelectedDevice") then
        tResponse["DATA"] = "<List><item><name>Local
        A</name><id>A</id></item><item><name>Local
        B</name><id>B</id></item><item><name>Local
        C</name><id>C</id></item></List>"
        SendToProxy(idBinding, "DATA_RECEIVED", tResponse)
```

Note the argument that exists between a service and device. Each has its respective command that is fired based on type. Both build a list a list of three lines of data which sent to the proxy using DATA_RECEIVED. These are three lines of data on each screen displayed in the screen samples above. The only difference is the string labels used to designate a "stream" for a streaming device and "Local" used for a device s that stress the media locally.

When an item is selected from the list on either of the screens the default XML command is fired. Let's look at the Lua code that is executed when a selection is made for in the HW Streaming Service driver:

```
elseif (strCommand == "GoToNowPlaying_StreamingService") then
                    tParams = ParseProxyCommandArgs(tParams)
                    G_nID = tParams.ARGS.id
                    G_sNAME = tParams.ARGS.name

                    --select song to play
                    tParams["ROOM_ID"] = tParams["ROOMID"]
                    tParams["STATION_URL"] = gMediaPath .. G_nID .. ".mp3"
                    tParams["QUEUE_INFO"] = G_nID
                    SendToProxy(G_nUI_PROXY_ID, "SELECT_INTERNET_RADIO", tParams, "COMMAND")


                    --update room media
                    tParams["LINE1"]     = "Line 1 (" .. G_sNAME .. ")"
                    tParams["LINE2"]     = "Line 2 (" .. G_sNAME .. ")"
                    tParams["LINE3"]     = "Line 3 (" .. G_sNAME .. ")"
                    tParams["LINE4"]     = "Line 4 (" .. G_sNAME .. ")"
                    tParams["IMAGEURL"]  = gMediaPath .. G_nID .. ".png"
                    tParams["MEDIATYPE"] = "secondary"
                    tParams["MERGE"] = "False"
                    SendToProxy(G_nUI_PROXY_ID, "UPDATE_MEDIA_INFO", tParams, "COMMAND", true)


                    --send UI to Now Playing screen
                    tParams["DATA"] = "<NextScreen>#nowplaying</NextScreen>"
                    SendToProxy(G_nUI_PROXY_ID, "DATA_RECEIVED", tParams)
```

We can see in the code that if the result of the argument is a streaming device driver it uses the SELECT_INTERNET_RADIO command and the UPDATE_MEDIA_INFO command to get data streaming. It is sent using the DATA_RECEIVED command.

Let's compare this model with how the HW Selected Device handles a selection from its screen. Here is the code from the driver:

```
elseif (strCommand == "GoToNowPlaying_SelectedDevice") then
                        tParams = ParseProxyCommandArgs(tParams)
                        G_nID = tParams.ARGS.id
                        G_sNAME = tParams.ARGS.name

                        tParams["ROOM_ID"] = tParams["ROOMID"]
                        SendToProxy(G_nUI_PROXY_ID, "SELECT_DEVICE", tParams, "COMMAND")

                        --update the queue with the song that we just selected to play
                        tParams["NAME"] = "QueueChanged"
                        tParams["EVTARGS"] =
                        "<NowPlaying>0</NowPlaying><List><item><Id>1</Id><name>" .. "Line 1 (" ..
                        G_sNAME .. ")" .."</name></item></List>"
                        SendToProxy(G_nUI_PROXY_ID, "SEND_EVENT", tParams, "COMMAND")

                        --update room media
                        tParams["LINE1"]     = "Line 1 (" .. G_sNAME .. ")"
                        tParams["LINE2"]     = "Line 2 (" .. G_sNAME .. ")"
                        tParams["LINE3"]     = "Line 3 (" .. G_sNAME .. ")"
                        tParams["LINE4"]     = "Line 4 (" .. G_sNAME .. ")"
                        tParams["IMAGEURL"]  = C4:Base64Encode(gMediaPath .. G_nID .. ".png")
                        tParams["MEDIATYPE"] = "secondary"
                        tParams["MERGE"] = "False"
                        SendToProxy(G_nUI_PROXY_ID, "UPDATE_MEDIA_INFO", tParams, "COMMAND", true)


                        --send UI to Now Playing screen
                        tParams["DATA"] = "<NextScreen>#nowplaying</NextScreen>"
                        SendToProxy(G_nUI_PROXY_ID, "DATA_RECEIVED", tParams)
```

In the HW Selected Device driver, we can see where SELECT_DEVICE is issued to select the actual device the driver supports. Next, a QueueChanged event is sent containing the data that was selected. It is sent using the SEND_EVENT Command. We also update the room information with same data information that was sent in the event: (`G_sName..Line 1`).

One point that needs to be understood is the need to use the Control4 API Base64Encode to get the image URL to display on the screen for a device type driver. The URL must be Base 64 encoded.

Once again we can see where the SELECT_INTERNET_RADIO command and the UPDATE_MEDIA_INFO command are used to get data streaming. This is sent using the DATA_RECEIVED command.

Additional Tools

Using the Schema Validator Utility

The Media Service Proxy Schema Validator is an application delivered by Control4 to assist with driver development. Specifically, it compares the XML found in the driver against the schema defined in the Media Service Proxy. It is a C# application that is compatible with the mono framework which allows the application to function across multiple platforms.

For more information on the mono software platform or to download the appropriate mono runtimes for your environment, go to: www.mono-project.com

Installing the Media Service Proxy Schema Validator

Locate the mspcv.zip file delivered with the Media Service Proxy. You will find this under the Utilities directory. Extract the mspcv.zip to your local, driver development environment.

Downloaded the mono runtime for your environment based on platform and version.

Schema Validator is a command line utility. You will need the current version of the MSP schema and the file that contains your driver's xml. In most cases this will be in the driver.c4z file.

Execute Schema Validator from the command line followed by the MSP schema file and the driver.xml file. For example:

```
SchemaValidator.exe schema.xsd driver.xml
```

Validation

The Media Service Proxy Schema Validator performs two types of schema validation. The first is a driver-wide validation of well-formed XML code. This validation focuses only on the XML structure of the code. It does not focus on the elements included in all of the driver's code.

For example, in the sample code below we see a malformed XML element in line 2. `<foo` should be `<foo/>`.

```
60.      <UI proxybindingid="5001"xmlns:xsi=
61.            "http://www.w3.org/2001/XMLSche ma-instance">
62.            <foo
               <Icons>
63.               <IconGroup id="Settings">
```

After running the Media Service Proxy Schema Validator, the following output is provided for this section of code:

```
Error parsing driver: expected '>' (3E) but found '<' (3C)
file:///Volumes/HD2/Users/testuser/Documents/C4/DriverWorksDrivers/Debug/MspNotifications/driver.xml Line 61, position 8.
```

The first line indicates what the application expected to see and what it actually found.

The next line details where the malformed XML is found. This includes the directory path to the XML and as well as the location within the code. In the example output above, "Line 61" refers to the sixty first line of the driver and position 8 refers to the position within that line.

The second type of validation the application performs is focused specifically on the UI section of the driver's Capability area. As described above, it will find any malformed XML code. In addition to this, the application will validate the XML code elements with the Media Service Proxy schema.

For example, in the sample code below, Line 61 contains a UI element not defined in the Media Proxy Schema: `<foo/>`

```
60.      <UI proxybindingid="5001"xmlns:xsi=
61.      "http://www.w3.org/2001/XMLSche ma-instance">
62.           <foo/>
         <Icons>
63.           <IconGroup id="Settings">
```

After running the Media Service Proxy Schema Validator, the following output is provided for this section of code:

```
Line 61 Pos 4, XmlSchema error: Invalid start element: :foo XML  Line 2,
Position 4.:   <foo />
```

The output begins with driver's line of code and the position of the invalid element. Next, it provides a schema error listing the element. Finally, it provides the line number of the Capabilities/UI XML code where the element is found. This is followed by its position within that line.

Media Service Proxy Create PO Utility

The Media Service Proxy Create PO Utility assists with the creation of .po files for the use in localizing the UI elements containing text that comes through the Media Service Proxy. The utility consists of a Python script called mspcreatpo.py. The script goes through all of the .lua files and extracts text strings as well as XML elements containing text that are defined in the Media Services Proxy schema. Using the utility will help with the often tedious and error prone manual creation of a .po file.

The Python Script used to create PO files for localization is called: mspcreatepo.py
If you need more information about installing Python or running Python scripts please see: http://www.python.org/

As mentioned above, the utility will look for text strings in the driver code. Specifically, it will look for strings wrapped in the gettext function and surrounded by double quotes. The gettext function is defined as:

```
function gettext(str)
  return str
end
```

For example, in the line of code below the strings Item 1 and item 1a would be extracted by the utility:

```
    {title=gettext("Item 1"), subtitle=gettext("item 1a"), icon="C4",
```

Let's look at a more detailed example of how the utility will work for an entire screen. Below is the XML for a List Screen. In order to localize this screen, and ready it for the utility you can see that we wrapped the text elements with the gettext method:

```
130.    function NavCmds.getTestListScreenItems(tParams)
131.      local items =
132.      {
133.    {title=gettext("Section 1"), isHeader="true", translate=true},
134.    {title=gettext("Item 1"), subtitle=gettext("item 1a"), icon="C4",
        length="2:48", isLink="true", translate=true},
135.    {title=gettext("Item 2"), subtitle=gettext("item 2a"), icon="C4",
        length="2:48", isLink="true", translate=true},
136.    {title=gettext("Section 2"), isHeader="true", translate=true},
137.        {title=gettext("Item 3"), subtitle=gettext("item 3a"),
        icon="C4",  length="2:48", isLink="true", translate=true},
138.    {title=gettext("Item 4"), subtitle=gettext("item 4a"), icon="C4",
        length="2:48", isLink="true", translate=true},
139.    {title=gettext("Section 3"), isHeader="true",   translate=true},
140.    {title=gettext("Item 5"), subtitle=gettext("item 5a"),
        icon="C4", length="2:48", isLink="true", translate=true},
141.    {title=gettext("Item 6"), subtitle=gettext("item 6a"),
        icon="C4", length="2:48", isLink="true", translate=true},
142.    {title=gettext("Item 7"), subtitle=gettext("item 7a"),
        icon="C4", length="2:48", isLink="true", translate=true}
143.      }
144.      local list = string.format("<List>%s</List>",
        arrayToXML(items))
145.      _sendData(tParams, list, nil)
```

```
146.     end
```

After running the mspcreatepo.py script, we have an output like this:

```
line: 133 Section 1
line: 134 Item 1
line: 134 item 1a
line: 135 Item 2
line: 135 item 2a
line: 136 Section 2
line: 137 Item 3
line: 137 item 3a
line: 138 Item 4
line: 138 item 4a
line: 139 Section 3
line: 140 Item 5
line: 140 item 5a
line: 141 Item 6
line: 141 item 6a
line: 142 Item 7
line: 142 item 7a
```

The utility also knows all of the XML UI elements defined by the Media Service Proxy schema which can be translated. When it runs, it will extract those elements. For example, in the Screens and Supporting Schema section we outlined the entire Setting Screen XML. All of the defined schema UI elements from that XML code block would be extracted by the utility. We would have an output like this:

```
Finding Settings Screen Labels:
Toggle Button 1:
Toggle Button 2:
Check Box 1
Check Box 2
ComboBox 1
ComboBox 2
Display Text:
Text 1
Text 2
Spinner 1 (0 to 1, inc .1)
Spinner 2 (1 to 10, inc 2)
```

Once the .po file has been generated by the utility, it can be translated into as many languages as needed. After translation, a capability called navigator_display_option is used to identify the location of .po files which may be displayed on the UI. The capability is defined as follows:

```
<navigator_display_option proxybindingid="5001">
   <translation_url>controller://driver/[DRIVER_NAME]/translations</translation_url>
</navigator_display_option>
```

Note the translation URL path that is used by the capability. This path points to the location of the .po file or in some cases .po files. For example,

driverFolder/www/translations/en_US.po
/en_UK.po
/es_CO.po

If numerous .po files are present, the .po file is selected based on the physical location of the device. If the device is mobile, it will use a .po that corresponds to whatever location the mobile device is currently recognizing.

The mscreatepo utility has a usage/help screen available from within it. Simply enter:

```
python mspcreatepo.py –h
```

This will display the help output as well as a list of parameters that can be pass into the utility.

Appendix

Media Service Proxy Commands

Proxy Commands are functions that are initiated from the user interface or they might be initiated by the Media Service Proxy itself. These commands are sent from (or through) the Media Service Proxy to the media device, service or cloud.

BackCommand
In OS release 2.9.0 and later, a new OPTIONAL top level UI XML item called: BackCommand. has been delivered. BackCommand is called when the back button is pressed. If the command is present its response will determine how the MSP behaves on a back press.

Possible responses are:
Empty (""): Proceed as a normal back press.
ReplaceScreen: Replace the current screen with a given screen id or xml screen definition.

GetDashBoard
Command to request the dashboard controls for a queue. This command should trigger a DashboardChanged Event from within the driver.

Parameters

NAVID – String parameter that represents a Navigator Device ID. The device id is used so that Navigator can filter out replies from other Navigators devices. Some Navigator devices might not have a driver. In that case a unique id will be generated.

LOCALE - Optional string parameter that represents the geographic locale of the Navigator device. For example, "en-US". If locale is not passed, the driver should use the system locale value.

SEQ - Integer parameter that is a Navigator-generated sequence ID value. The ID value is useful when trying to match correct replies with Navigator requests.

ROOMID - Integer parameter that is the ID value of the room where the Navigator is located

GetQueue
Command that triggers a QueueChanged Event. This is called when a Navigator loads the MSP's Now Playing Screen. A QueueChanged event should be sent to the Navigator who called GetQueue.

Parameters

NAVID – String parameter that represents a Navigator Device ID. The device id is used so that Navigator can filter out replies from other Navigators devices. Some Navigator devices might not have a driver. In that case a unique id will be generated.

LOCALE - Optional string parameter that represents the geographic locale of the Navigator device. For example, "en-US". If locale is not passed, the driver should use the system locale value.

SEQ - Integer parameter that is a Navigator-generated sequence ID value. The ID value is useful when trying to match correct replies with Navigator requests.

ROOMID - Integer parameter that is the ID value of the room where the Navigator is located

GET_HAS_DISCRETE_MUTE
Command requesting the ability of the media service or device to support discreet mute.

Parameters
None

GET_HAS_DISCRETE_VOLUME
Command requesting the ability of the media service or device to support discreet volume

Parameters
None

GET_VOLUME_LEVEL
Command requesting the current volume level from the media device or service.

Parameters
None

INTERNET_RADIO_SELECTED
Called when an URL has been selected to play by Digital Audio.

Parameters
QUEUE_ID - The Id of the queue the command was initiated by.
QUEUE_INFO - User defined queue data.
ROOM_ID - The id of the room where the URL was selected.
STATION_URL - The URL that was selected to play.

DEVICE_SELECTED
Notification sent to the protocol whenever a device is selected in a room.

Parameters
location: STRING: a location, e.g. url, ID or other media service specific string identifying the location of idMedia. This is optional. Whenever this “location” is presented (not nil), the media is selected.
idRoom: INT: The room selecting the device.
idMedia: The mediaId associated with the media selected
mediaType: If media is being selected, the type of media.

Example

```
function ReceivedFromProxy(idBinding, strCommand, tParams)
     if (strCommand == "DEVICE_SELECTED") then
          if (tParams["location"] ~= nil) then
               print("ReceivedFromProxy(): DEVICE with location " ..
tParams["location"])
          end
     end
end
```

DEVICE_DESELECTED
Notification sent to the protocol when a device is no longer selected in a room.

Parameters
idRoom: INT: The room that the deselected device is in.

MUTE_OFF
Turn muting off.

Parameters
OUTPUT:(INT)nOutputBindingID

MUTE_ON
Turn muting on.

Parameters
OUTPUT (INT)nOutputBindingID

MUTE_TOGGLE
Toggle muting on/off.

Parameters
OUTPUT: (INT)nOutputBindingID

PLAY
Called when the PLAY command is executed from within a room. This will place the media device or service in a playback state

Parameters
ROOM_ID - ID of the room where the command originated.

PAUSE
Called when the PAUSE command is executed from within a room. This will place the media device or service in a pause state

Parameters
ROOM_ID - ID of the room where the command originated.

PULSE_VOL_DOWN
Pulse volume level down.

Parameters
OUTPUT:(INT)nOutputBindingID

PULSE_VOL_UP
Start ramping volume.

Parameters
OUTPUT: (INT)nOutputBindingID

QUEUE_DELETED
Indicates the queue has been removed from Digital Audio.

Parameters
QUEUE_ID - The Id of the queue the command was initiated by.
QUEUE_INFO - User defined queue data.
LAST_STATE_TIME - The amount of time in seconds the queue was in the last state
LAST_STATE - The last state the queue was in before being deleted. See QUEUE_STATE_CHANGED STATE parameter for a list of valid states.

ROOMS - A list of rooms in the zone.

QUEUE_MEDIA_INFO_UPDATED
Sent when a queues media info has changed

Parameters
QUEUE_ID - The Id of the queue the command was initiated by.
QUEUE_INFO - User defined queue data.
MEDIA_INFO:

```
<mediainfo>
    <deviceid>100002</deviceid>
    <queueid>10019</queueid>
    <streamed>4095</streamid>
    <source>5</source>
    <mediatype>SONG</mediatype>
    <mediatypeV2>GENERIC_MEDIA</mediatypeV2>
    <medSrcDev>25</medSrcDev>
    <stationid>[URL]</stationid>
    <queueInfo>0</queueInfo>
    <album>Stereolithic</album>
    <artist>311</artist>
    <title>Ebb and Flow</title>
    <channel/>
    <img>aHR0cDovL3N0YXRpYy5yaGFwLmNvbS9pbWcvNjAweDYwMC8xL
    zQvMS8zLzUxMDMxNDFfNjAweDYwMC5qcGc=</img>
</mediainfo>
```

QUEUE_STATE_CHANGED
Sent from proxy when Digital Audio is the selected device

Parameters

QUEUE_ID - The Id of the queue the command was initiated by.
QUEUE_INFO - User defined queue data.
PREV_STATE_TIME - Amount of time in seconds the queue was in the previous state.
PREV_STATE - The previous queue state. See STATE for possible values.
STATE - The current state of the queue, possible values are:
PLAY - Queue is currently playing
PAUSE - Queue is paused
STOP - The queue has been stopped
END - No more items to play in the queue

RESELECT_SOURCE
Command sent to the device owning the currently selected media when a path re-selection of media source occurs. This allows the device to detect a change in the media.

Parameters
PATH_TYPE
MEDIA_ID
ROOM_ID

SELECT_DEVICE
Command that takes a ROOM_ID parameter that is the room id containingthe device Generally, this should be the same room the Navigator sends to the driver via ReceivedFromProxy->tParams->ROOMID parameter. Navigator will only send ROOMIDs that have a valid path to the device.

Parameters
ROOM_ID (INT) – RoomID value that contains the device.

SELECT_SOURCE
Command sent to the device owning the currently selected media when a path selection of media source occurs. This allows the device to detect a change in the media.

Parameters
PATH_TYPE
MEDIA_ID
ROOM_ID

SET_VOLUME_LEVEL
Change volume value to a specified level.

Parameters
LEVEL: (INT)nLevel
OUTPUT:(INT)nOutputBindingID

SettingChanged
Command sent when a setting has changed. If the responses' DATA object contains Settings XML, the screen will be updated with those settings. Not all settings need to be returned. Zero or more settings are considered valid returns.

Parameters

NAVID – String parameter that represents a Navigator Device ID. The device id is used so that Navigator can filter out replies from other Navigators devices. Some Navigator devices might not have a driver. In that case a unique id will be generated.

LOCALE - Optional string parameter that represents the geographic locale of the Navigator device. For example, "en-US". If locale is not passed, the driver should use the system locale value.

SEQ - Integer parameter that is a Navigator-generated sequence ID value. The ID value is useful when trying to match correct replies with Navigator requests.

ROOMID - Integer parameter that is the ID value of the room where the Navigator is located

ARGS - Arguments for the command as defined in the UI XML which include
ScreenID – The ID of the Settings screen where the setting was changed.
PropertyName – String parameter for the name of the property that was changed.
Value – The new value of the property.

Responses:
Can be one of the following:
Empty (""): Do nothing.
NextScreen: Push the next screen on to the view stack.
ReplaceScreen: Replace the current screen.
RemoveScreen: Removes the current screen going back to the previous screen.
Settings: Update some or all of the screen's values.
RefreshScreen: Re-issue original DataCommand to driver.

SKIP_FWD
Called when the SKIP_FWD command is executed from within a room. This will cause the media device or service to skip forward through media.

Parameters:
ROOM_ID - Id of the room where the command was sent from

SKIP_REV
Called when the SKIP_FWD command is executed from within a room. This will cause the media device or service to skip backwards through media.

Parameters:
ROOM_ID - Id of the room where the command was sent from

START_VOL_UP
Stop ramping volume.

Parameters
OUTPUT (INT)nOutputBindingID
START_VOL_DOWN

Begin ramping volume down.

Parameters
OUTPUT:(INT)nOutputBindingID

STOP
Called when the STOP command is executed from within a room. This will stop the media device or service from its previous playback state.

Parameters
ROOM_ID - ID of the room where the command originated.

STOP_VOL_DOWN
Begin ramping volume down.

Parameters
OUTPUT (INT)nOutputBindingID

STOP_VOL_UP
Stop ramping volume.

Parameters
OUTPUT:(INT)nOutputBindingID

Command Type Parameters used in Device Commands

The three commands listed in the previous section are pre-defined commands used by the Media Service Proxy. Numerous other device-specific commands will need to be created during the process of driver development. These commands will dictate how the UI and the driver communicate with each other. In order for the UI to get data (as requested by interaction from the end user) it must issue a device command to the device protocol. For example, every time the user presses a UI button a device command is associated with that button press. The command is evoked and sent to the Media Service Proxy which then passes the command on to the device protocol.

The amount and scope of this command set will vary based on the device or service the driver is being created to support. The following section will outline several examples of device commands. When reviewing these examples, it is important to understand that while device commands will be unique to any given device, the command's parameter type or `<Type></Type>` value is defined within the schema of the Media Service Proxy. The proxy expects one of seven pre-defined command parameter type values when a device command passes through it. The Media Service Proxy Command Parameter Type values are:

FIRST_SELECTED
DATA_OFFSET
DATA_PAGE
DATA_COUNT
DEFAULT
SEARCH
SEARCH_FILTER

What follows are examples of commands using each of the command type parameters listed above. In all of the examples, data is passed from the UI through the ReceivedFromProxy command in the form of a table of parameters. This table contains a parameter called Args or `tParams.ARGS`. This is a string of XML.

Example 1: Using the FIRST_SELECTED Command Type Parameter
The following example shows the use of a Default Command. Default Commands are evoked anytime a list item is pressed on the UI. The name of the Default Command in the example is "selectMenuItem." As named, the command is sent when a user presses a UI element found within a list of menu items. The command type is PROTOCOL. This is seen in line 3. This command type is used when the command is sent directly from the UI through the Media Service Proxy to the device protocol driver. The other type of supported command type is ROOM. Note that when ROOM is designated as command type the ARGS data is ignored as the command is sent to the Room Driver.

The use of the FIRST_SELECTED command parameter type can be seen in line 7 below. Note that multi-selecting items within the UI is not currently supported through the Media Service Proxy. FIRST_SELECTED refers to the selection of an item, not the first item selected in a group of selected item.

For example, a device driver sends the following XML to a UI device to display a Home button. That XML would look like the following:

```
<List>
   <item>
     <name>Go Home</name>
     <type>GoHome</type>
     <icon>Settings</icon>
   </item>
 </List>
```

When the user presses the "Home" button, the following command is executed.

```
1.    <DefaultCommand>
2.      <Name>selectMenuItem</Name>
3.      <Type>PROTOCOL</Type>
4.      <Params>
5.        <Param>
6.          <Name>type</Name>
7.          <Type>FIRST_SELECTED</Type>
8.          <Value>type</Value>
9.        </Param>
10.     </Params>
11.   </DefaultCommand>
```

As a result, the UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
Args: <args><arg name="type">GoHome</arg></args>
```

Example 2: Using the DATA_OFFSET Command Type Parameter
The following example shows the use of a Data Command. Data Commands differ from Default Commands in that they are used to request data from the driver with the purpose of displaying that data on the UI.

The name of the command is "getOffsetPaginationData". The command type is PROTOCOL meaning that the command is sent from the UI to the device protocol driver.

Line 7 shows the use of the DATA_OFFSET command parameter type. This command parameter type is used to set the pagination style used to display data on the UI. OFFSET supports the data being displayed in a page-by-page manner.

For example, consider a list of tracks displayed on a UI. When the user gets to the bottom of the page, a new page of data is loaded. The DATA_OFFSET command parameter type is zero-based in that a value of zero will return data starting with the first item of the list. A value of 5 would return a list of items starting with the fifth item at the top of the UI.

In order to use any "Data_" command parameter, the pagination style of the screen must be set to something other than NONE. In the example that follows, it is set to OFFSET.

When a screen is loaded, the following command is executed.

```
1.    <DataCommand>
2.        <Name>getOffsetPaginationData</Name>
3.        <Type>PROTOCOL</Type>
4.        <Params>
5.            <Param>
6.                <Name>start</Name>
7.                <Type>DATA_OFFSET</Type>
8.            </Param>
9.            <Param>
10.               <Name>count</Name>
11.               <Type>DATA_COUNT</Type> <!-- Number of items for driver to
                        return -->
12.           </Param>
13.       </Params>
14.   </DataCommand>
```

As a result, the UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
<args><arg name="start">0</arg><arg name="count">20</arg></args>
```

Note, in order to use any of the DATA_ parameters, the screen must support pagination.

Example 3: Using the DATA_PAGE Command Type Parameter
The following example shows the use of a Data Command. Data Commands differ from Default Commands in that they are used to request data from the driver with the purpose of displaying that data on the UI.

The name of the command is "getPagePaginationData". The command type is PROTOCOL meaning that the command is sent from the UI to the device protocol driver.

Line 7 shows the use of the DATA_PAGE command parameter type. This command parameter type is used to set the data page number to display on the UI. The PAGE command parameter type is one-based in that a value of one will return data items from the first page of data, and a value of 2 would return a list of items starting with the first item on the second page of data.

In order to use any "Data_" command parameter, the pagination style of the screen must be set to something other than NONE. In the example that follows, it is set to PAGE.

When a screen is loaded, the following command is executed.

```
1.      <<DataCommand>
2.          <Name>getPagePaginationData</Name>
3.          <Type>PROTOCOL</Type>
4.          <Params>
5.              <Param>
6.                  <Name>page</Name>
7.                  <Type>DATA_PAGE</Type> <!-- 1 based page number to load ->
8.              </Param>
9.              <Param>
10.                 <Name>count</Name>
11.                 <Type>DATA_COUNT</Type> <!-- Number of items per page -->
12.             </Param>
13.         </Params>
14.     </DataCommand>
```

As a result, the UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
<args><arg name="page">1</arg><arg name="count">20</arg></args>
```

Note, in order to use any of the DATA_ parameters, the screen must support pagination.

Example 4: Using the DATA_COUNT Command Type Parameter
The following is example of using the DATA_COUNT Command Parameter Type comes from the previous code sample used to explain DATA_PAGE. Line 11 shows the DATA_COUNT command parameter type in the code. This value represents the number of items that will be displayed within the confines of one page on the UI.

When a screen is loaded, the following command is executed.

```
1. <DataCommand>
2.     <Name>getPagePaginationData</Name>
3.     <Type>PROTOCOL</Type>
4.     <Params>
5.         <Param>
6.             <Name>page</Name>
7.             <Type>DATA_PAGE</Type> <!-- 1 based page number to load -->
8.         </Param>
9.         <Param>
10.             <Name>count</Name>
11.             <Type>DATA_COUNT</Type> <!-- Number of items per page -->
12.              </Param>
13.          </Params>
14.      </DataCommand>
```

As a result, the UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
<args><arg name="page">1</arg><arg name="count">20</arg></args>
```

Note, in order to use any of the DATA_ parameters, the screen must support pagination.

Example 5: Using the DEFAULT Command Type Parameter
The DEFAULT Command Parameter Type is used when the value of the parameter is meant to be passed to the driver "as is". This means that the value will not be changed or modified by the UI.

For example, when a user presses a button called "Not Good" the following command is execute:

```
1.    <Button>
2.      <Name>Not Good</Name>
3.      <Command>
4.        <Name>ButtonPressed</Name>
5.        <Type>PROTOCOL</Type>
6.        <Params>
7.            <Param>
8.              <Name>type</Name>
9.              <Type>DEFAULT</Type> <!-- Hardcoded value in the UI xml -->
10.             <Value>Not Good</Value> <!-- Taken as is -->
11.           </Param>
12.       </Params>
13.     </Command>
14.   </Button>
```

As a result, the UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
<args><arg name="type">Not Good</arg></args>
```

Example 6: Using the SEARCH Command Type Parameter
The SEARCH Command Type Parameter only applies to Data Command types. It is used when a Data Command is expecting a search query based on the input from the user.

For example, consider that a user searches media based on an artist such as "U2." The user will perform a search in the UI for "U2" and click on the Search Button. When the Search Button is pressed, the following command is executed:

```
1.   <DataCommand>
2.     <Name>GetBrowseMenu</Name>
3.     <Type>PROTOCOL</Type>
4.     <Params>
5.       <Param>
6.         <Name>search</Name>
7.         <Type>SEARCH</Type>
8.         <!-- This parameter will contain the search query if the user is
             searching for something -->
9.       </Param>
10.      <Param>
11.        <Name>search_filter</Name>
12.        <Type>SEARCH_FILTER</Type>
13.        <!-- This parameter will contain the search filter Id if the user
             is searching for  something -->
14.      </Param>
15.    </Params>
```

As a result, the UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
<args><arg name="search">U2</arg><arg name="search_filter">Artist<arg></args>
```

Example 7: Using the SEARCH_FILTER Command Type Parameter
The following example is another Data Command. The name of the command is "getBrowseMenu" and the command type is PROTOCOL.

The SEARCH_FILTER Command Type Parameter can be seen in line 11. This Command Type Parameter contains the Search Filter ID value used to retrieve filtered data when the user searches.

```
1.    <DataCommand>
2.      <Name>GetBrowseMenu</Name>
3.      <Type>PROTOCOL</Type>
4.      <Params>
5.        <Param>
6.          <Name>search</Name>
7.          <Type>SEARCH</Type>
8.        </Param>
9.        <Param>
10.         <Name>search_filter</Name>
11.         <Type>SEARCH_FILTER</Type>
12.       </Param>
13.     </Params>
```

The UI will send the following Args to the Media Service Proxy and then on to the protocol:

```
<args><arg name="search">U2</arg><arg name="search_filter">Artist<arg></args>
```

Media Service Protocol Notifications

Protocol Notifications are responses sent from the device protocol due to a Proxy Command being received.

DATA_RECEIVED
Called by protocol when responding to a DeviceCmd. The Media Service Proxy will forward the response to Navigator in a dataToUI

Parameters

NAVID – String parameter that represents a Navigator Device ID. The device id is used so that Navigator can filter out replies from other Navigators devices. Some Navigator devices might not have a driver. In that case a unique id will be generated.

SEQ - Integer parameter that is a Navigator-generated sequence ID value. The ID value is useful when trying to match correct replies with Navigator requests.

DATA - XML - The data being forwarded from the protocol to the UI

ERROR – Optional string parameter for the error message to be displayed. If this parameter is passed, an error has occurred.

SEND_EVENT
Called by a protocol when it needs to send an asynchronous event to a navigator device. If neither NAVID, nor ROOMS is provided, the event will be broadcast to all navigators.

Parameters

NAVID – String parameter that represents a Navigator Device ID. The device id is used so that Navigator can filter out replies from other Navigators devices. Some Navigator devices might not have a driver. In that case a unique id will be generated.

ROOMS – Optional parameter of a list of comma-separated room ID values for navigators that care about this event. This parameter is ignored if the NAVID parameter is provided.

NAME – String parameter for the name of the event

EVTARGS – XML List of parameters send within the Notification.

Note: An Event is ignored if a NAVID is sent to a Room where that Navigator device does not exist. It will also be ignored if a Room ID is sent along with a NavID that does not exist within that room. Control4 recommends that either NAVID or ROOMs are used – not both.

UPDATE_MEDIA_INFO
Command that the protocol sends to the proxy whenever the now playing information changes.

Parameters

LINE1 – STRING
LINE2 – STRING
LINE3 – STRING
LINE4 – STRING
IMAGEURL – STRING: The URL of the image.
ROOMID – NUMBER: The id of the room the info is playing in
MERGE – BOOLEAN: True or False –This parameter dictates how data that has previously been pass through the UPDATE_MEDIA_INFO command is handled. If the MERGE argument is set to True, any optional arguments omitted will leave the existing data untouched. They will be merged in with the new data passed in the command.

If the MERGE argument is set to False, any optional arguments omitted will clear the existing data. Regardless of the value of the MERGE argument, any empty strings provided in the optional arguments will clear the data.

For example, consider that UPDATE_MEDIA_INFO was previously used to set the following fields:
LINE1 – STRING: Artist Information
LINE2 – STRING: Album Information
LINE3 – STRING: Track Information
IMAGEURL – STRING: URL of the Album Cover

When The UPDATE_MEDIA_INFO command is evoked again, a new URL for the album cover is passed. If the Merge parameter is set to True – the three other parameters will be passed with their previous data. If the UPDATE_MEDIA_INFO command is evoked in the same scenario as above, but with the Merge parameter set to False – only the new IMAGEURL parameter will be passed. The three other parameters will be cleared.

SELECT_INTERNET_RADIO
Command that the protocol sends to the proxy in order to play a URL. Even though the name is "INTERNET_RADIO" this command is used to play all supported audio formats through a URL.

Parameters

ROOM_ID – Integer parameter for the ID of the room where internet radio will play.
STATION_URL – String parameter for the URL of the audio item to play. This does not have to be a radio station

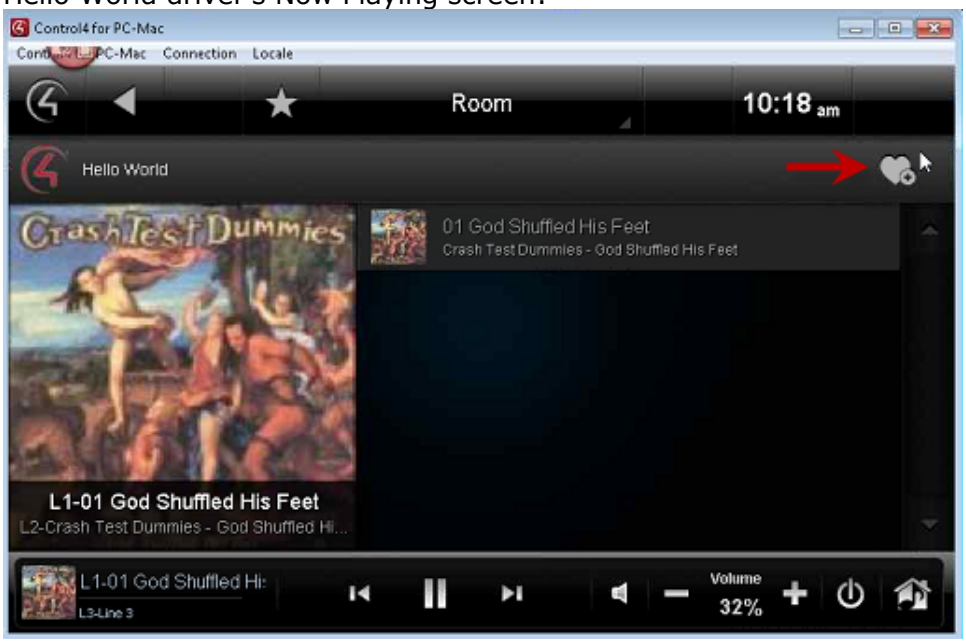
QUEUE_INFO – String parameter that is pass-through context data, which will be placed onto the audio queue. For example, it could be the original URL used to play the zone or the current playlist id, etc.

VOLUME – Optional. If present will cause the room's or audio zone's volume to be set to that level. This argument should only be set by the driver if this call was triggered by a DEVICE_SELECTED notification that supplied a volume.

Media Service Proxy Properties

ActionIdsProperty

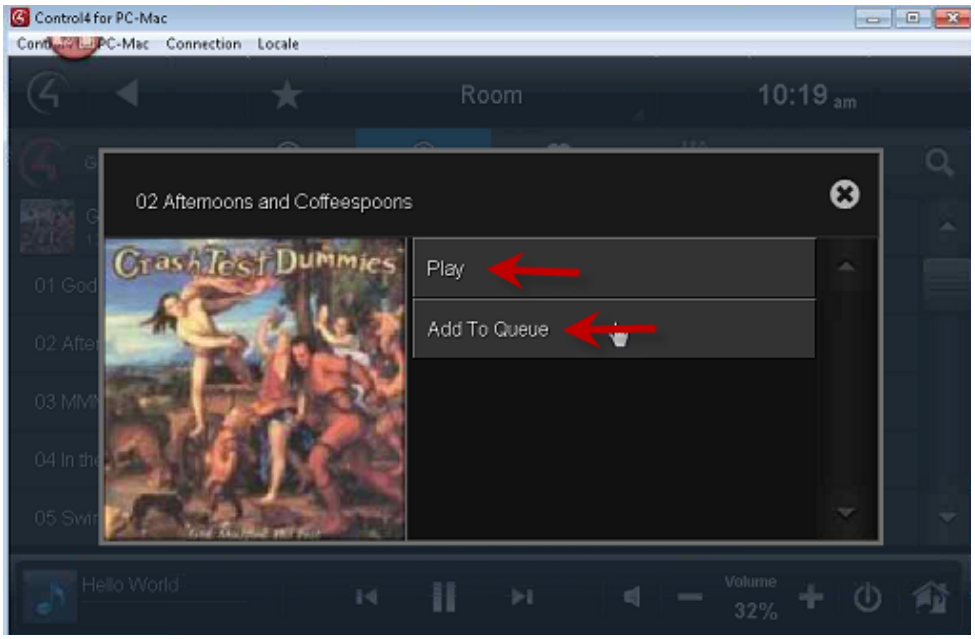
A new property called ActionIdsProperty has been added to indicate the property of the NowPlaying or Collection screen's data that indicates the current set of actions that should be displayed. For example, these are Actions are displayed at the Collection level in the Hello World driver's Now Playing screen:



In the example above, the "Add to Favorites" icon is displayed in the NowPlaying screen. This icon is representative of an Action declared using the ActionIdsProperty. The name of the Action is called "Add to Favorites" and its Action ID is "Preset". The Action ID can be referenced in the driver's Lua code and can dynamically change the Action displayed on the screen based on queue content. The Preset Action has a Filter associated with it. Filters are associated with items returned in the queue and used to show a state of an Action. Depending on the items returned in the queue in the Hello World driver, Filters dictate if the "Add to Favorites" or the "Remove from Favorites" icon is displayed. Filters also dictate if no icons appear. For example, if the item in the queue is not a song or a station, no icons are displayed. For code samples, please review the ActionIdsProperty and Preset XML for the Hello World driver.

ItemActionIdsProperty

A new property called ItemActionIdsProperty has been added that indicates which item's property contains a list of Actions. The Actions are used in place of the list's global Actions. For example, these are Actions are displayed at the list item level in the Hello World driver's Now Playing screen:



This property has been added to resolve the issue where Action Filters were being incorrectly used to support lists assembled from different data sets. Action Filters are intended to show item state – such as if an item was favored or not. For example, when an item is pressed, if the item has the property defined by ItemActionIdsProperty and the property has 1 or more valid actions are listed: the action popup with the referenced actions are displayed. If no actions are valid or none are given then the default action will be invoked. The default action will not show a popup.

ItemDefaultActionProperty

A new list element called ItemDefaultActionProperty has been added. It indicates which item's property references the action used as the default for that item. At runtime, if an item has the property, that property will override the DefaultAction for that list item or collection. For example, when an item is pressed and the item has the property defined by ItemDefaultActionProperty, this action is ran. If the item does not have the property defined by ItemDefaultActionProperty then DefaultAction is ran. An example of this can be seen in the Hello World driver's Browse Stations screen. An XML excerpt from that screen looks like this:

```
<DefaultAction>BrowseStations</DefaultAction>
<ItemDefaultActionProperty>actionIds</ItemDefaultActionProperty>
```

Since an ItemDefaultActionProperty is defined (actionIds), this Action is run instead of the DefaultAction of BrowseStations when the user selects an Item from the Browse Station screen. This only happens when and Item has an actionIds parameter defined. The actionId property used here is defined as the PlayStations Action. It fires a command called PlayStationsCommand.

This supports the ability to have a Default Action when a screen loads and then other Actions can be fired when a user selects UI elements at the item level.

Media Service Proxy Capabilities

<digital_audio_support>True</digital_audio_support>
Capability designating whether or not the media device or service can play digital audio

<has_discrete_volume_control>True</has_discrete_volume_control>
Capability designating whether or not the media device or service has discreet volume control

<has_discrete_mute_control>True</has_discrete_mute_control>
Capability designating whether or not the media device or service has discreet mute control

<has_up_down_volume_control>False</has_up_down_volume_control>

<can_scan_media>False</can_scan_media>
Capability designating whether or not the media device or service has the ability to scan media

<translation_url></translation_url></navigator_display_option>
Indicates where the localization .PO files are located. Everything after DRIVER_NAME is assumed to be in the www folder of the .c4z driver. An example of the URL passed into this capability would be: `controller://driver/DRIVER_NAME/[WWW_PATH_TO_PO_FILES]`

<hide_in_media>True</hide_in_media>
Capbillity designating whether or not the driver should be hidden from Composer Pro's Media tab.

<ui_selects_device>true</ui_selects_device>
Capability that dictates the result of pressing the media device's icon in Navigator. If set to True, the device will be set as the room's selected device.

<list_nowplaying_non_c4>True</ list_nowplaying_non_c4>true>
Capability that indicates an MSP driver that does not use Control4's DIGITAL_AUDIO but uses Non-Digital Audio instead. Setting this capability to True will allow for media metadata and transport controls to be displayed on devices such as the Neeo remote.

<list_no_nowplaying>False </list_no_nowplaying>
Capability that indicates an MSP driver that should never automatically jump to the Now Playing screen when a favorite is selected.

Media Service Proxy Events

Media Service Proxy Events are sent from the driver to the user interface. Note that these are different that "Events" found in ComposerPro. Navigators receive the Media Service Proxy Event's XML. If the NAVID tag is present, only the Navigator with the matching ID will process this message. If the ROOMS tag is present, Navigators will only process this message if they're in one of the rooms listed with in this tag.

DashboardChanged
This event is sent by a protocol when the dashboard controls should be changed.

Parameters

Items – STRING: A space separated string with a list of Id values from a Transport command, which is defined in the driver.xml Dashboard section.

Dashboard section in driver.xml:
Each Transport command requires an ID, ButtonType and ReleaseCommand.
Valid Transport ButtonType values are: PLAY, PAUSE, STOP, SKIP_REV, SKIP_FWD and CUSTOM. CUSTOM values additionally require a unique Name and Icon parameter. The Icon parameter should reference an IconGroup from the Icons section. For example:

```
<Dashboard>
                <Transport>
                    <Id>Play</Id>
                    <ButtonType>PLAY</ButtonType>
                    <ReleaseCommand>
                        <Name>PLAY</Name>
                        <Type>ROOM</Type>
                    </ReleaseCommand>
                </Transport>
                <Transport>
                    <Id>Pause</Id>
                    <ButtonType>PAUSE</ButtonType>
                    <ReleaseCommand>
                        <Name>PAUSE</Name>
                        <Type>ROOM</Type>
                    </ReleaseCommand>
                </Transport>
                <Transport>
                    <Id>Stop</Id>
                    <ButtonType>STOP</ButtonType>
                    <ReleaseCommand>
                        <Name>STOP</Name>
                        <Type>ROOM</Type>
                    </ReleaseCommand>
                </Transport>
                <Transport>
                    <Id>SkipRev</Id>
                    <ButtonType>SKIP_REV</ButtonType>
                    <ReleaseCommand>
                        <Name>SKIP_REV</Name>
                        <Type>PROTOCOL</Type>
                    </ReleaseCommand>
                </Transport>
                <Transport>
                    <Id>SkipFwd</Id>
                    <ButtonType>SKIP_FWD</ButtonType>
                    <ReleaseCommand>
                        <Name>SKIP_FWD</Name>
                        <Type>PROTOCOL</Type>
                    </ReleaseCommand>
```

```
        </Transport>
        <Transport>
            <Id>ShuffleOn</Id>
            <ButtonType>CUSTOM</ButtonType>
            <Name>Turn Shuffle On</Name>
            <IconId>amt_shuffle</IconId>
            <ReleaseCommand>
                <Name>ToggleShuffle</Name>
                <Type>PROTOCOL</Type>
            </ReleaseCommand>
        </Transport>
        <Transport>
            <Id>ShuffleOff</Id>
            <ButtonType>CUSTOM</ButtonType>
            <Name>Turn Shuffle Off</Name>
            <IconId>amt_shuffle_on</IconId>
            <ReleaseCommand>
                <Name>ToggleShuffle</Name>
                <Type>PROTOCOL</Type>
            </ReleaseCommand>
        </Transport>
        <Transport>
            <Id>RepeatOn</Id>
            <ButtonType>CUSTOM</ButtonType>
            <Name>Turn Repeat On</Name>
            <IconId>amt_repeat</IconId>
            <ReleaseCommand>
                <Name>ToggleRepeat</Name>
                <Type>PROTOCOL</Type>
            </ReleaseCommand>
        </Transport>
        <Transport>
            <Id>RepeatOff</Id>
            <ButtonType>CUSTOM</ButtonType>
            <Name>Turn Repeat Off</Name>
            <IconId>amt_repeat_on</IconId>
            <ReleaseCommand>
                <Name>ToggleRepeat</Name>
                <Type>PROTOCOL</Type>
            </ReleaseCommand>
        </Transport>
    </Dashboard>
```

QueueChanged

This event is sent by a protocol when the queue list has changed and should be refreshed.

Parameters

List – XML: The queue list that should be displayed.

NowPlaying – Optional. This supports global Now Playing actions. Previously, Now Playing actions were tied to the currently playing track. This new element prevents a full queue update when changing a now playing action. The element contains a series of key, value pairs that are used as inputs to the Now Playing actions. The NowPlaying List element will no longer affect the global NowPlaying actions. The NowPlaying element's values will serve as input to the Now Playing actions for filtering and command parameters. If an Action is called using <NowPlaying> and that Action has a command associated with it – the <NowPlaying> element needs to package all of the Action's parameters. Note: "key" can be any element name similar to List items. For example, here is the <NowPlaying> element using the Preset Action. Note the inclusion of the Preset command's parameters:

```
local NowPlaying = "<NowPlaying><actionIds>Preset</actionIds>" ..
```

```
            "<key>" .. gNowPlaying[gCurrentSongIndex].key .. "</key>" ..
            "<type>" .. gNowPlaying[gCurrentSongIndex].type .. "</type>" ..
            "<is_preset>" .. gNowPlaying[gCurrentSongIndex].is_preset ..
            "</is_preset>" ..
        "</NowPlaying>"
```

QueueChangedevent

Parameters

Id – STRING: The id of the item that has changed. The UI will map this item to the queue's id property that was mapped in the UI spec.

Note: If something has been added or removed from the queue, use the QueueChanged event.

ProgressChanged
This event is sent by a protocol when the dashboard status should be updated. The driver should broadcast this event asynchronously. If you know which rooms are in the media zone, you can send the event just to those rooms. Otherwise, you can send them to all UIs and the UIs will figure out if they need the information or not. This event should not be sent more frequently than once per second.

Parameters

offset (optional) INTEGER: If this parameter is present, the "length" parameter must also be present. This value represents the current position in the stream between 0 and "length".
length (optional) INTEGER If this parameter is present, the "offset" parameter must also be present. This value represents the total length of the stream.
buffer (optional) INTEGER : If this parameter is present, the system is currently buffering. The value is a percentage value in the range of 0..100. If this parameter is not present, the buffer is considered full (the system is playing), or no buffering is needed.
label (optional) STRING An optional label to display next to a progress bar.

DriverNotification
Displays a driver notification dialog on the UI.

Parameters

Id - STRING: The id of the driver notification dialog to display (described in the UI).
Title – STRING: The title of the dialog to display.
Message – STRING: The main dialog message to display.
Image (optional) – STRING: URL for image to display. Multiple Image tags can be present. If multiple are present, a width and height attribute should be present on each tag so that the best image can be shown for each display device.
Context (optional) - OBJECT: Pass driver specific context information to the notification. If the notification sends a command back to the driver (from the user pressing a button) then the context object will be passed as the data.

Media Service Proxy Variables

UI System Variables:
Beginning with OS 2.9.0, Media Service Proxy Drivers now have access to some UI system variables. This is supported with the delivery of a new parameter type of "SYSTEM". Predefined system variables will be sent to the driver by defining a command parameter's type as SYSTEM
and its value as the name of the system variable. For example:

```
<Param>
      <Name>entryPoint</Name>
      <Type>SYSTEM</Type>
      <Value>menu</Value>
</Param>
```

The example above would send a command with the parameter named "entryPoint" that
will have the possible 3 values: "", "listen", "watch"

Note that the because of the SYSTEM parameter type, the MENU parameter has been deprecated and replaced with the "menu" system variable

System Variables:
screenDepth
Also delivered with operating system 2.9.0 is a new screenDepth system variable The screenDepth variable will be the number of screens currently on the view stack. For example if
the view stack is: "Tab Root" -> "Child 1" -> "Child 2". The screenDepth variable will return 3.

screenID
A new screenId system variable has been delivered with OS release 2.9.0. The screenId variable will be the id of the current top screen on the view stack.

tabID
A new tabId system variable has been delivered with OS release 2.9.0. The tabId variable will return the id of the current tab selected in the UI.
In previous releases, some services require handling screen navigation and UI view stacks manually.

Other:
PLAYING_AUDIO_DEVICE
This variable tracks the currently playing audio device and allows for programming actions in the room when the device playing the audio changes. If programming that detects when a room's audio source changes exists, that programming will not run if this variable is not set.

Media Service Proxy Driver Connections

This section is contains two examples of Driver Connection. The first example is from a digital audio service or cloud. The second is from a physical media device.

Media Service/Cloud Example:

```
<connections>
  <connection>
    <id>5001</id>
    <facing>6</facing>
    <connectionname>Media Service</connectionname>
    <type>2</type>
    <consumer>False</consumer>
    <audiosource>False</audiosource>
    <videosource>False</videosource>
    <linelevel>False</linelevel>
    <classes>
      <class>
        <classname>MediaService</classname>
      </class>
    </classes>
  </connection>
  <connection>
    <id>3001</id>
    <facing>6</facing>
    <connectionname>Digital Audio Client</connectionname>
    <type>6</type>
    <consumer>True</consumer>
    <audiosource>False</audiosource>
    <videosource>False</videosource>
    <linelevel>True</linelevel>
    <classes>
      <class>
        <autobind>True</autobind>
        <classname>DIGITAL_AUDIO_CLIENT</classname>
      </class>
    </classes>
  </connection>
  <connection>
    <id>4100</id>
    <facing>6</facing>
    <connectionname>Digital Audio</connectionname>
    <type>6</type>
    <consumer>False</consumer>
    <audiosource>True</audiosource>
    <videosource>False</videosource>
    <linelevel>True</linelevel>
    <classes>
      <class>
        <autobind>True</autobind>
        <classname>DIGITAL_AUDIO_SERVER</classname>
      </class>
    </classes>
  </connection>
</connections>
```

Physical Device Example:

```
<connections>
  <connection>
    <id>5001</id>
    <facing>6</facing>
    <connectionname>MusicBridge</connectionname>
    <type>2</type>
    <consumer>False</consumer>
    <audiosource>True</audiosource>
    <videosource>False</videosource>
    <linelevel>False</linelevel>
    <classes>
      <class>
        <classname>MediaService</classname>
      </class>
    </classes>
  </connection>
  <connection>
    <id>4000</id>
    <facing>6</facing>
    <connectionname>AUDIO OUT</connectionname>
    <type>6</type>
    <consumer>False</consumer>
    <audiosource>True</audiosource>
    <videosource>False</videosource>
    <linelevel>True</linelevel>
    <classes>
      <class>
        <classname>STEREO</classname>
      </class>
      <class>
        <classname>DIGITAL_COAX</classname>
      </class>
    </classes>
  </connection>
  <connection>
    <id>7000</id>
    <facing>6</facing>
    <connectionname>AUDIO OUT End-Point</connectionname>
    <type>7</type>
    <consumer>False</consumer>
    <audiosource>False</audiosource>
    <videosource>False</videosource>
    <linelevel>False</linelevel>
    <classes>
      <class>
        <classname>AUDIO_SELECTION</classname>
      </class>
      <class>
        <classname>AUDIO_VOLUME</classname>
      </class>
    </classes>
  </connection>
</connections>
```

Media Service Proxy Driver Notification Prototype

Prototype Example:

```
<Notification>
  <Id>GoHome</Id>
     <Buttons>
        <Button>
           <Name>By Command</Name>
           <Command>
             <Name>GoHome</Name> <!— Returns a NextScreen response with #home —>
             <Type>PROTOCOL</Type>
           </Command>
        </Button>
        <Button>
           <Name>By Screen Id</Name>
           <ScreenId>#home</ScreenId>
         </Button>
     </Buttons>
 </Notification>
```

Notification Code Example:

```
local context = {
          foo = 'bar',
          baz = 'bash',
 }

mAddService',
          InstanceId = self.NAVID,
          Title = getText ('Add Service'),
          Message = getText ('Press OK to add ') .. XMLEncode (NAME) .. getText (' to your project'),
          Context = context
 }

 SendEvent (5001, nil, nil, 'DriverNotification', params)
```

Receiving the Context in the Callback

```
 function Navigator:ConfirmAddServiceOK (idBinding, seq, args)
          local context = JSON:decode (C4:Base64Decode (args.context)) or {}

          local foo = context.foo
          local baz = context.baz
```

Notification In XML format

```
<Notification>
          <Id>ConfirmAddService</Id>
          <CancelButton>
                    <Name>Cancel</Name>
                    <Command>
                              <Name>ConfirmAddServiceCancel</Name>
                              <Type>PROTOCOL</Type>
                    </Command>
          </CancelButton>
          <Buttons>
                    <Button>
                    <Name>OK</Name>
                    <Command>
                              <Name>ConfirmAddServiceOK</Name>
                              <Type>PROTOCOL</Type>
                              <Params>
                                        <Param>
                                                  <Name>context</Name>
                                                  <Type>FIRST_SELECTED</Type>
                                                  <Value>context</Value>
                                        </Param>
                              </Params>
                    </Command>
          </Button>
          <Button>
                    <Name>Cancel</Name>
                    <Command>
                              <Name>ConfirmAddServiceCancel</Name>
                              <Type>PROTOCOL</Type>
                              <Params>
                                        <Param>
                                                  <Name>context</Name>
                                                  <Type>FIRST_SELECTED</Type>
                                                  <Value>context</Value>
                                        </Param>
                              </Params>
                    </Command>
          </Button>
          </Buttons>
 </Notification>
```

Media Service Proxy Pagination

The pagination type called ALPHA supports displaying data lists that have alpha numeric indexes. For example, some services provide a list of letters/digits and an index tied to each letter/digit. This map of letter/digits to indexes is known in the MSP as an "AlphaMap". For example an AlphaMap would be something like:

```
Letter     Index
---------------
5            0
9           34
硅          56
B           78
C           100
```

A screen implementing this type will return from its data command a <AlphaMap> element, in addition to other standard elements like <List>. The AlphaMap element will contain a list of<item> elements with a <key> and <index> element. The key element will contain the human readable string that begins at the index provided in the index element. The indexes MUST be in order and NOT duplicated. Unicode MUST also be supported for proper localization. UIs will show the Alpha map in their platform dependent way.

Requesting data will use the same mechanism as the OFFSET pagination type. Also, the DataCommand response SHOULD contain a "length" attribute within its List element like:

```
<List length="300">
```

For example an AlphaMap would look like:

```
<AlphaMap>
        <item>
                <key>0</key>
                <index>0</index>
        </item>
        <item>
                <key>1</key>
                <index>25</index>
        </item>
        <item>
                <key>5</key>
                <index>75</index>
        </item>
        <item>
                <key>A</key>
                <index>83</index>
        </item>
        <item>
                <key>B</key>
                <index>209</index>
        </item>
        <item>
                <key>硅</key>
                <index>256</index>
        </item>
</AlphaMap>
```

The use of the PaginationStyle XMl tag dicyates which style is used for each screen. The style types are:
NONE or `<PaginationStyale>NONE</PaginationStyle>`

OFFSET or `<PaginationStyale>OFFSET</PaginationStyle>`
PAGE or `<PaginationStyale>PAGE</PaginationStyle>`
The following is an example of a List type screen using the OFFSET PaginationStyle:

```
<Screens>
<Screen type="ListScreenType">
	<Id>LibraryListScreen</Id>
	<PaginationStyle>OFFSET</PaginationStyle>
	<List>
		<ItemDefaultActionProperty>defaction</ItemDefaultActionProperty>
		ItemActionIdsProperty>actions</ItemActionIdsProperty>
		<TitleProperty>title</TitleProperty>
		<SubTitleProperty>creator</SubTitleProperty>
		<ImageProperty>albumArtURI</ImageProperty>
		<IsLink>
			<Property>canEnumerate</Property>
			<ValidValues>
				<Value>true</Value>
			</ValidValues>
		</IsLink>
	</List>
	<DataCommand>
		<Name>LibraryBrowseCommand</Name>
		<Type>PROTOCOL</Type>
		<Params>
			<Param>
				<Name>id</Name>
				<Type>FIRST_SELECTED</Type>
				<Value>id</Value>
			</Param>
			<Param>
				<Name>start</Name>
				<Type>DATA_OFFSET</Type>
			</Param>
			<Param>
				<Name>count</Name>
				<Type>DATA_COUNT</Type>
			</Param>
			<Param>
				<Name>search</Name>
				<Type>SEARCH</Type>
			</Param>
			<Param>
				<Name>filter</Name>
				<Type>SEARCH_FILTER</Type>
			</Param>
		</Params>
	</DataCommand>
</Screen>
```

Understanding Icons and their Resolutions

The Media Service Proxy recognizes several types of icons which are displayed on user interfaces. These include: Device Icons, Branding Icons as well as Tab, Action and List Icons. This section will focus on Device and Branding Icons as these two icons are supported from the base class of the Media Service Proxy.

Device Icons are user interface elements that are used in Control4's UI Listen and Watch menu. It is assumed that all Icons are square. Typically, Device Icons represent a driver for a device or service. In the example below there are three Device Icons displayed representing drivers for TuneIn®, The Wireless Music Bridge and Rhapsody®.

A Media Service Driver developer must provide custom Device Icons that meet a defined set of resolutions. This is in order to ensure a consistent user interface experience across the numerous devices that run Navigator. Required resolutions for Device Icons are as follows:

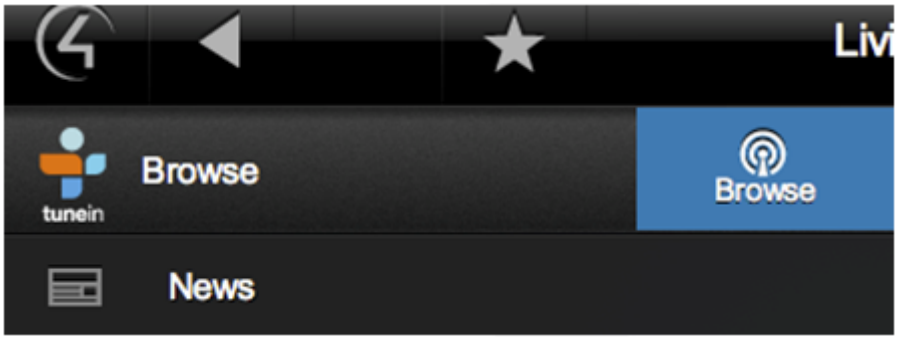
"25x25"
“30x30"
"40x40"
"50x50"
"56x56"
"60x60"
“70x70"
"80x80"
"90x90"
"100x100"
"110x110"
"120x120"
“130x130"
“140x140”
“300x300"

Note: This list represents all currently supported Media Service Proxy resolutions. Future versions of this document will identify Device Icon specific resolutions. These values will likely change.

Control4 recommends providing a resolution from 20x20 px to 140x140 px in 10 px increments. Navigators are guaranteed to pull an icon on a 10 px boundary (10, 20, 30, etc..). If the icon at the preferred resolution is not found, the next closest resolution will be pulled with a preference for larger icons. The icon will be scaled to fit the space.

Cover art is not guaranteed to be on a 10 pixel boundary and will be scaled (maintaining aspect ratio) to fit the space.

While Branding Icons can appear identical to Device Icons their purpose and use by the Media Service Proxy differs. A Branding Icon is a user interface element that is displayed on a Media Service Proxy page and is used to identify the page. In the example below, the TuneIn® icon is used to identify the TuneIn user interface screen.



Just like Device Icons, Branding Icons must be provided for a defined set of resolutions. Those resolutions are:

"25x25"
“30x30"
"40x40"
"50x50"
"56x56"
"60x60"
“70x70"
"80x80"
"90x90"
"100x100"
"110x110"
"120x120"
“130x130"
“140x140”
“300x300"

Note: This list represents all currently supported Media Service Proxy resolutions. Future versions of this document will identify Branding Icon specific resolutions. These values will likely change.

Icon Groups

IconGroups represent a single visual representation, for example "Browse" or "My Favorites". An Icon inside of the icon group is a single resolution of the IconGroup's visual representation. In essence, the IconGroup allows multiple resolutions to be defined. Navigators will choose to display the Icon that best matches the platform running on the Navigator. Where the icon is displayed also determines which resolution is selected.

The URL scheme that is used to reference the icons from within the driver is as follows:

```
controller://driver/[DRIVER_NAME]
```

To reference a drivers' www/ folder, the "controller://" syntax gets translated in Navigators to:

http://[CONTROLLER IP]/

Note that the [DRIVER_NAME] syntax is the file name of the driver without the .c4z. file extension.

Control4 recommends driver development using the Media Service Proxy occur in conjunction with the c4z driver framework. While driver creation using the c4i format is possibly, it is not supported with the Media Service Proxy.

For example, an individual icon in the TuneIn driver is referenced as:

```
controller://driver/TuneIn/icons/40x40/act_log_tunein.png
```

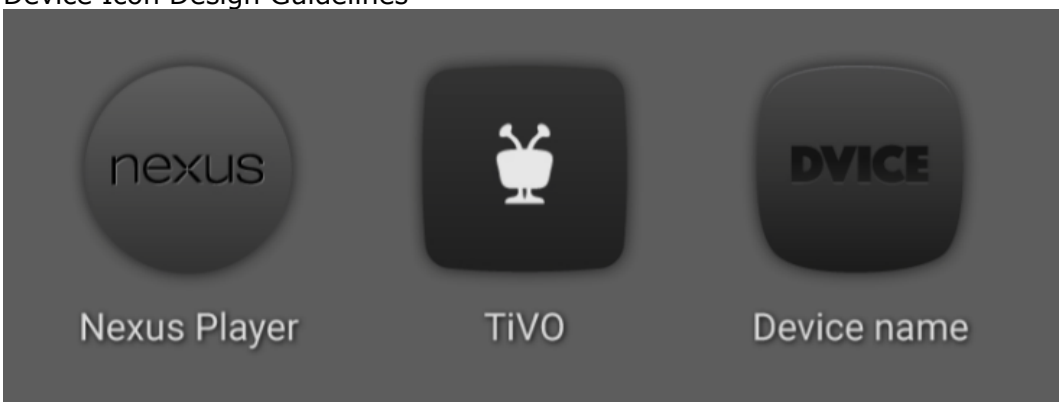
The example above uses icons folder found within the TuneIn c4z driver.

Creating Media Service Proxy Compatible Icons

Devices and Services Icons
This section describes the creation of two types of icons: Device Icons and Service Icons. Before continuing, it is important to understand the differences between a "Device" and a "Service" in the Control4's Media Service Proxy architecture.

- Devices are physical components of equipment integrated into a Control4 project.
- Services are experiences offered in the Control4 user interface through an experience or a connected device.

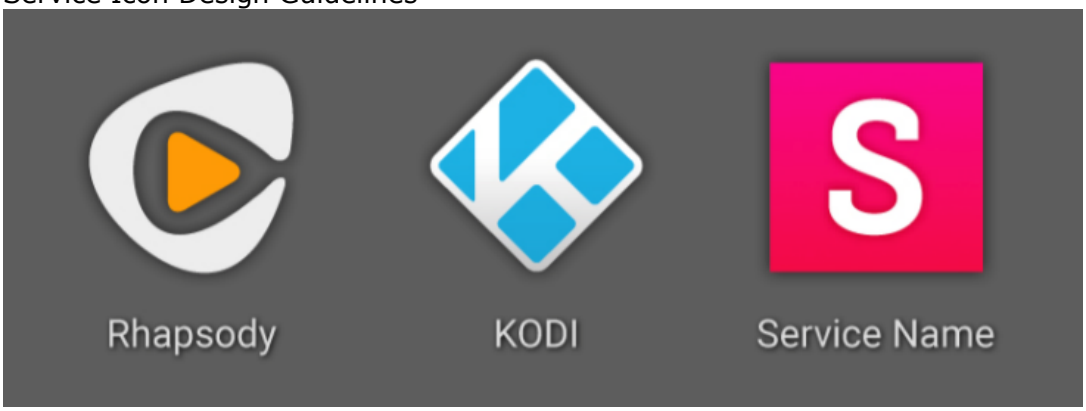
Device Icon Design Guidelines



Artwork for Device icons should be simplified and vector-based (not a photo) and represent the device’s most recognizable or signature shape and color(s). They may include the products' logo as it appears on the device.

Because the Device’s name will likely be displayed below the icon, placing additional text or logos on the device icon beyond what is actually on the physical device may be redundant and is discouraged.

Service Icon Design Guidelines



Artwork used for Service icons should have a simple, clear, app-like appearance. Usually displaying an icon or logo mark.

Service icons may be any shape. If an icon or logo is complex, a background shape may be used to improve visibility.

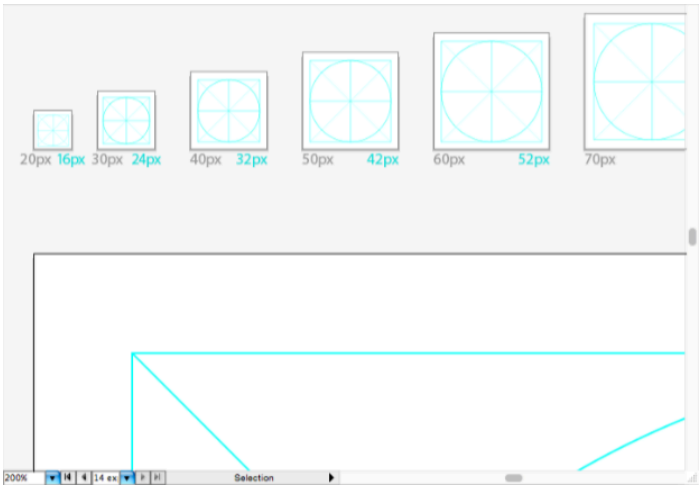
Because the Service's name will likely be displayed below the icon, placing the full name of the service on the icon graphic itself may be redundant and is discouraged.

Directions

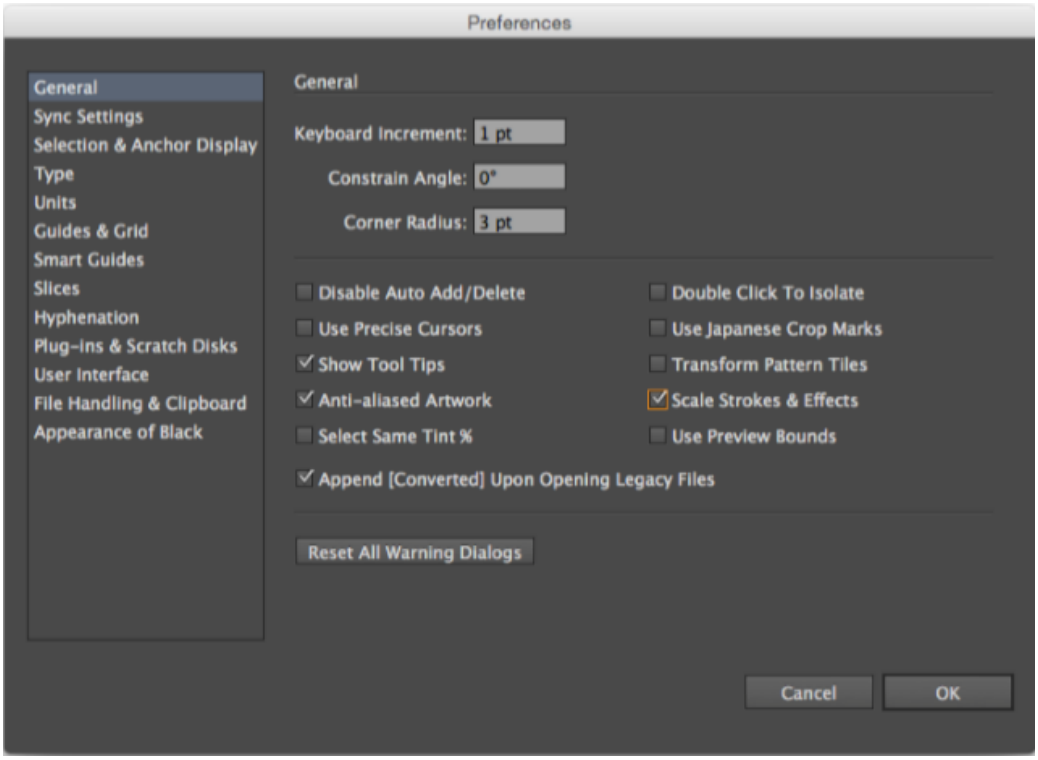
NOTE:
Before you begin, you may choose to hide placeholder graphics or toggle their visibility in the Illustrator Layers palette. (Window > Layers).

All resources for creating MSP icons are included in the DriverWorks SDK at the following location: DriverWorks SDK\Documentation\Media Service Proxy Resources\Icon Resources

1. Open the provided Illustrator template named "**Device & Serivce Icon Template.ai**"



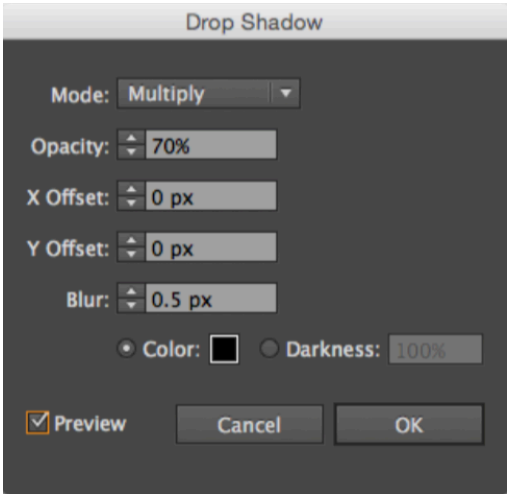
2. Open Illustrator Preferences and verify that ‘Scale Strokes & Effects’ option is checked.



3. Import your own artwork or create vector artwork in Illustrator template file.

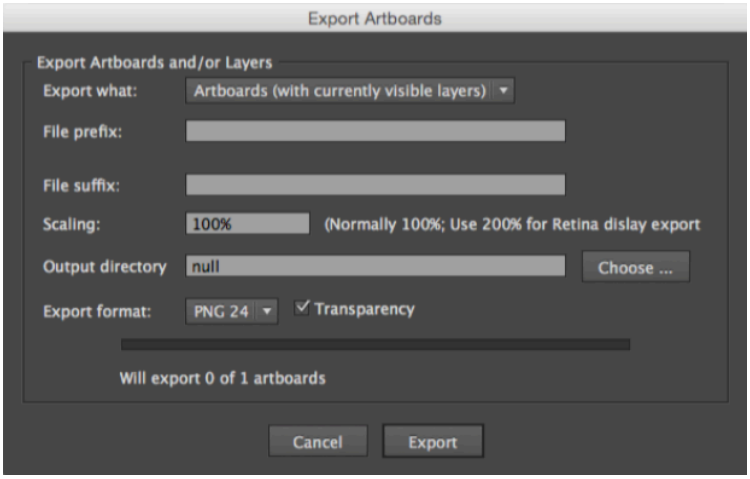
4. Resize your icon to fit inside 16px blue guides on the smallest (20px) artboard. Icon edges must stay within blue guides. All icons will be smaller than their artboards to accommodate shadow.

5. Apply the following drop shadow settings to 20px (16px) icon graphic. Shadows will fall outside blue guides must stay within artboard edges. All icons will be smaller than their artboards to accommodate shadow.



6. Copy, paste and resize existing 20px (16px) graphic with shadow to all larger sizes. DO NOT apply a new drop shadow to each graphic.

7. Export all the artboards at the same time to a folder using the free MultiExporter script available for download at: http://www.ericson.net/content/2011/06/export-illustrator-layers-andor-artboards-as-pngs- and-pdfs/



The MultiExporter script will export and name each of the Illustrator artboards as a separate .PNG file. The names for each of the .PNG files should be as follows:
experience_20.png
experience_30.png
experience_40.png
experience_50.png
experience_60.png
experience_70.png
experience_80.png
experience_90.png
experience_100.png
experience_110.png
experience_120.png
experience_130.png
experience_140.png
experince_300.png

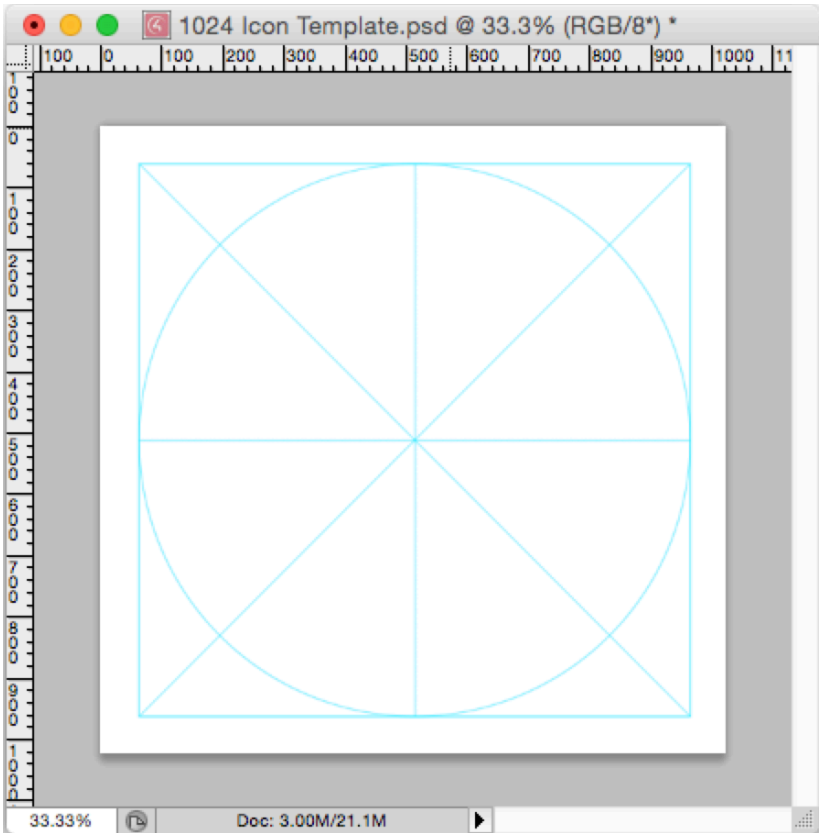
8. Verify that the icons appear correctly visually and the file names are correct.

9. Place the folder of .png icons into the www directory of your .c4z file.

Photoshop Directions

1. Open the provided Photoshop template file named "**Device & Service Icon Template.psd**"

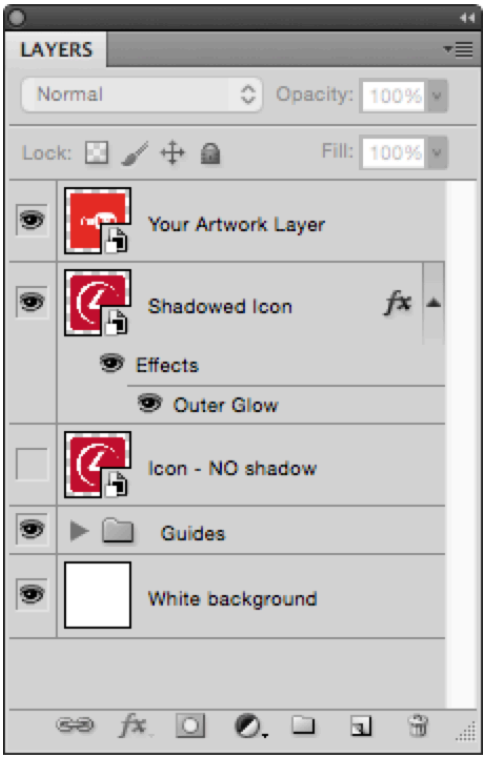
NOTE: At any time you may hide placeholder graphics by toggling their visibility in the Layers palette. (Window > Layers)



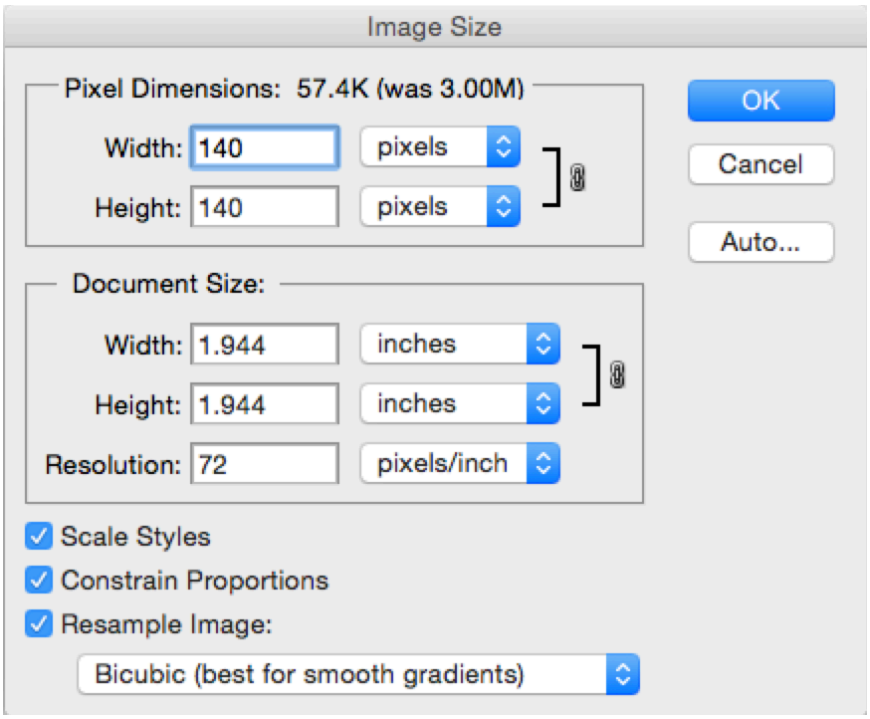
2. Import (or drag and drop) high-resolution artwork of at least 72dpi at 1024 pixels for your device or service icon and scale edges to outer blue guides.

3. Click on the Outer Glow effect layer and while holding down the option key, drag and drop the effect from the "Shadowed icon" layer to your own artwork layer. This will copy the correct Outer Glow effect. If you do not see the effects layer, click the small triangle next to the 'fx' to reveal Effects.

NOTE: If you have multiple layers, you will need to merge the layers before applying outer glow. Merge your layers by selecting each visible layer and selecting **'Layers' > 'Merge Layers'**



4. Select: '**Image' > 'Image Size'** to reveal the Image size dialog (below).
Resize the 1024px version to each of the smaller sizes in 10-pixel increments, beginning with 20 pixels and moving all the way to 140 pixels. Because the template is square, you will only need to enter a Width or Height. Do not change any other settings.



Use the following naming convention to create 14 unique files:

experience_20.png
experience_30.png
experience_40.png
experience_50.png
experience_60.png
experience_70.png
experience_80.png
experience_90.png
experience_100.png
experience_110.png
experience_120.png
experience_130.png
experience_140.png
experience_300.png

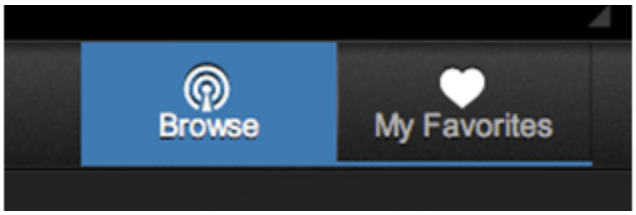
5. Place these 14 icons into a folder and verify the icons appear correctly visually and the file names are correct.

6. Place the folder of completed .png icons into the www directory of the c4z file

Using Tabs to Enhance Screen Navigation

Tabs are useful as they provide a means for the end user to navigate to a defined user interface screen. Tabs are root-level, user interface elements and consists of an icon and a text string representing each Tab. The Icon Group ID associates the Icon with the Tab.

In the example below, two Tabs are displayed. Each consists of an icon and name.



For example, the “Browse” icon is displayed along with the name: Browse.

Beginning with operating System 2.9.0, the use of multiple sets of Tabs are supported. There are instances when a different set of tabs should be presented to the user. For example, If Bluetooth is disabled on the Wireless Music Bridge a "Devices" tab should not be presented to the user.

To support the display of multiple Tab sets, a new Command element has been created inside of Tabs so that only a list of tabs or the command can be defined at the same time. The Tabs Command will be called when the browse portion of a driver is entered.

The Command will return <Tabs> xml with nested <Tab> values. The Tab elements will be in the format defined by the schema.

The tabs returned by the command are valid for the duration of a user session. A session is defined as the starting with when the user enters the driver to when the DESTROY_NAV is called.

Example XML:

```
<Tabs>
	<Command>
		<Name>GetTabs</Name>
		<Type>PROTOCOL</Type>
	</Command>
</Tabs>
```

Also included with operating system 2.9.0 is Tab XML enhancements to support dynamic screen functionality. Prior to 2.9.0 , Tabs were statically associated with a screen and therefore unable to participate in the dynamic screen functionality.

Beginning with 2.9.0, an OPTIONAL ScreenCommand inside the Tab XML definition has been provided. The ScreenCommand will be called when a tab is pressed. Its response will be a NextScreen response. Note that ScreenCommand is mutually exclusive with ScreenId.

An Overview of the .c4z File Format

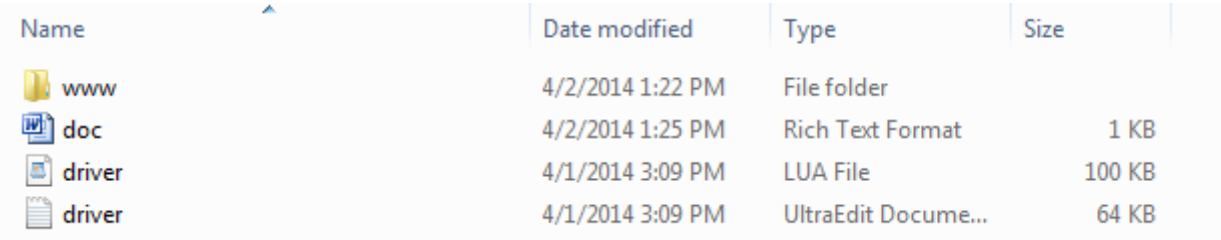
Beginning with operating system 2.6.0, Control4 has changed the file structure previously used to support device drivers. Up until 2.6.0, device driver were contained within a single-level file using the extension of .c4i or: `drivername.c4i`

The release of 2.6.0 marks a significant change to this model with the introduction of the .c4z file structure or: `drivername.c4z`. A .c4z file provides all of the content that the .c4i file previously did. However, it offers some significant features and advantages that its predecessor did not.

At its most basic level, a .c4z file is a zip file. It contains numerous folders and files which, when encapsulated in a .c4z file, represent a device driver. The modularity within the .c4z file makes for a far more organized and structured approach to device driver architecture. It also provides the ability to include many related objects within the confines of the driver in an organized manner.

These objects include items such as graphics and icon directories to support custom user interfaces which can run on Navigator. A .c4z file also contains a .lua file, which contains all of the driver's lua code. It has its own .xml directory as well to separate the XML portion of a driver from the .lua code. Also, rich text documentation can be included in the .c4z to support a far superior user assistance model.

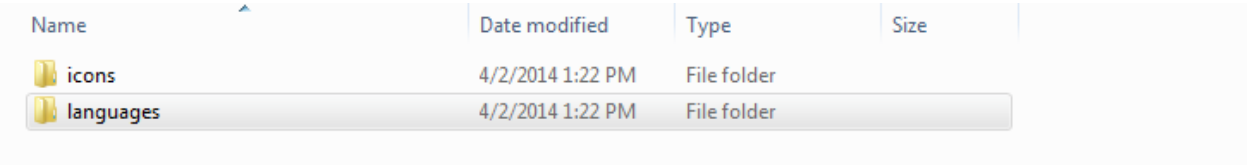
The image below is a look into a .c4z file at its root level:



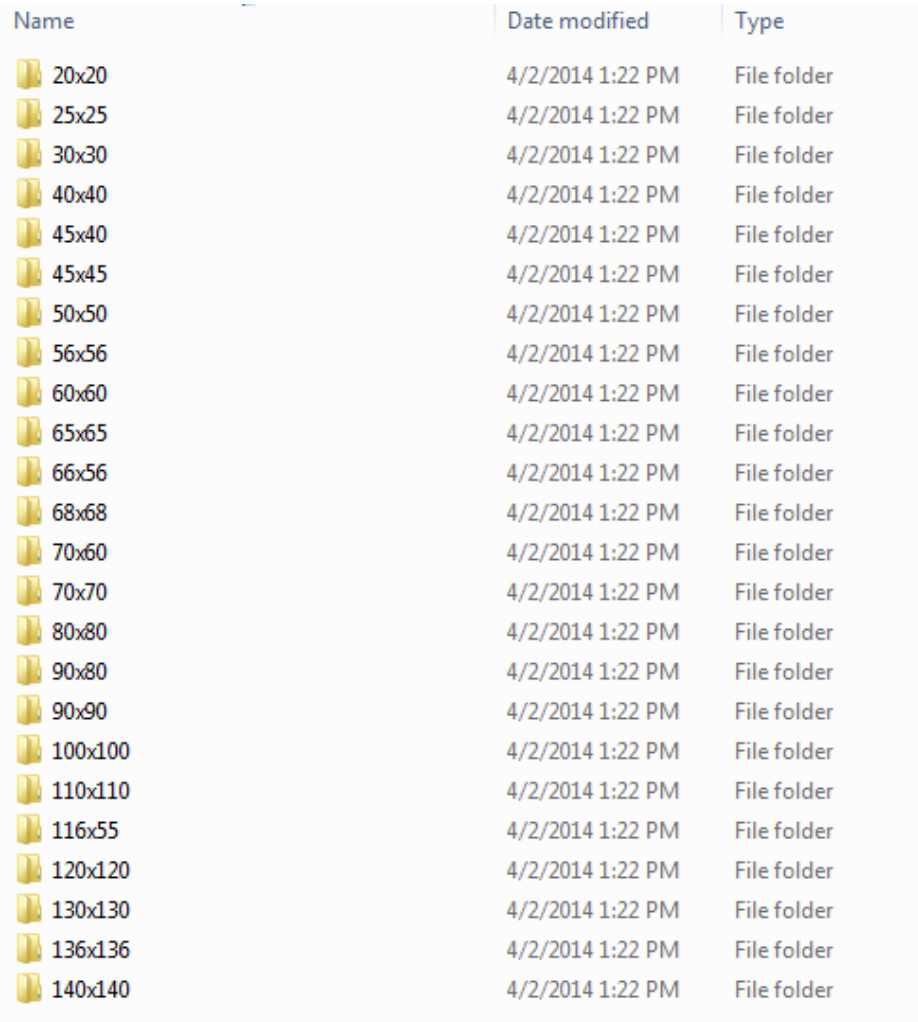
You'll notice a few files at the root level that were mentioned above. The root level contains the documentation file to support the driver. Below that is the .lua file that contains all of the lua code for the driver.

Next we can see another driver file. This contains all of the XML that was previously found between the `<devicedata>` tags of a .c4i file. These are elements such as `<creator>`, `<name>`, `<model>`, `<manufacturer>` and so on.

If we open the www directory we'll see the following:



As mentioned above, a c4z file can contain graphical elements to support the driver's use in Navigator. When opened, the icons folder for this driver looks like this:



The icons directory contains all of the images, organized by their resolutions, which are displayed during the use of the driver. The images in this particular driver are found under the root level of "www" placing them within the .c4z in this manner makes them accessible

from via the controller's webserver. For example, accessing an image can be accomplished by appending the .c4z icon path to a URL such as:

http://urlstring/driver/drivername/icons/20x20/driverimage.png

or

http://127.0.0.1/driver/myc4zdriver/icons/20X20/driverimage.png

The other directory file found under this driver's www directory is the languages folder. This folder contains all of the .po files used for localizing this driver.

Going forward, any .lua-based driver will be expected to be delivered in the .c4z format. To facilitate the conversion of previously built .c4i files to the new .c4z format, control4 has delivered a utility called DriverPackager. DriverPackager accomplishes two significant tasks for the developer. These include:

- XML Validation
- Assembles the .c4z

DriverPackager is access through the DriverEditor tool.

Creating Browsable Lists of Media Elements

The ability of a Media Service Proxy driver to deliver a browse-able list of media related elements has been included. These lists may represent station, channels or actual media. The elements delivered in the list can then be used as parameter in a command.

Initially, a command needs to be created in the devicedata/config section of the c4i/c4z driver.xml file. For example, here is a command called SelectChannel:

```
<devicedata>
	…
	<config>
		…
		<commands>
			<command>
				<name>SelectChannel</name>
				<description>Select channel PARAM1</description>
				<params>
				   <param>
					<name>Channel</name>
					<type>CUSTOM_SELECT:SelectChannelParamSelect</type>
				   </param>
				</params>
			</command>
		</commands>
	</config>
	…
</devicedata>
```

In the SelectChannel example above, SelectChannel is the command name that is sent to the ExecuteCommand function in the driver. The new param type CUSTOM_SELECT allows for the specification of a global .lua function that will be invoked whenever ComposerPro wants to get data. In the example above, it is the function SelectChannelParamSelect.

The command description can also contain variables enclosed in {{ and }}, and the name refers to the param name.
Note, that this variable substitution currently only works with params of type CUSTOM_SELECT.

Next, the newly created .lua function needs to be implemented. The simplest implementation just returns a list of strings:

```
function SelectChannelParamSelect(currentValue)
	local list = {}
	for i = 1, 10 do
		table.insert(list, "Channel " .. i)
	end
	return list
end
```

It is possible to return human-readable text, while keeping internal values hidden (e.g. to display a station's name rather than its station id). Keep in mind that these values must be of type string:

```
function SelectChannelParamSelect(currentValue)
	local list = {}
	for i = 1, 10 do
		table.insert(list, { text = "Channel " .. i, value = tostring(i) })
	end
	return list
end
```

You can also create a browsing experience with folders. Note, that the function can return an arbitrary string value as second return value. If this occurs a "Back" link with that value will be displayed. This example also allows selecting the "Most Popular" folder (by default folders are not selectable):

```
function SelectChannelParamSelect(currentValue)
	local back = nil
	local list = {}
	if (string.sub(currentValue, 1, 7) == "popular") then
	     back = "" -- Back to the root menu
	     for i = 1, 10 do
	     table.insert(list, { text = "Channel #" .. i, value = "popular" .. i })
	       end
	elseif (string.sub(currentValue, 1, 9) == "playlists") then
	    back = "" -- Back to the root menu
	    table.insert(list, { text = "Mine", value = "myplaylists", folder = true })
	    for i = 1, 5 do
	    table.insert(list, { text = "Playlist " .. i, value = "playlists" .. i })
	    end
	elseif (string.sub(currentValue, 1, 11) == "myplaylists") then
	    back = "playlists" -- Back to the Playlists
	    for i = 1, 2 do
	    table.insert(list, { text = "My playlist " .. i, value = "myplaylists" .. i })
	    end
	elseif (string.sub(currentValue, 1, 9) == "favorites") then
	    back = "" -- Back to the root menu
	    for i = 1, 3 do
	    table.insert(list, { text = "Favorite " .. i, value = "favorites" .. i })
	    end
	else
	    table.insert(list, { text = "Playlists", value = "playlists", folder = true })
	    table.insert(list, { text = "Favorites", value = "favorites", folder = true })
	    table.insert(list, { text = "Most popular", value = "popular", folder = true,
	        selectable = true })
	       for i = 1, 5 do
	       table.insert(list, { text = "Channel " .. i, value = "channel" .. i })
	       end
	end

	return list, back
end
```

You can also defer returning the list. This may be useful if you are waiting for the completion of a web service call or waiting for data from a device to return. The function you're implementing (SelectChannel) is receiving a function pointer as a second argument, which can be saved off and can be called to return the list at a later time. This function either takes the list and back value, or just a string if ComposerPro should display an error message.

In order to defer returning data, you would not return anything from the function, but then when you have the data, call the provided function. Note that if you fail to call this provided function, and also don't return anything from your function, then Composer will display an error message. Also, you may only call this function once. Here's an example:

```
g_tickets = {}

function ReceivedAsync(ticketId, strData, responseCode, tHeaders, strError)
	local ticket = g_tickets[ticketId]
	if (ticket ~= nil) then
		g_tickets[ticketId] = nil

		if (strError ~= nil) then
			-- Display the error
			ticket.returnData("Web Service Error: " .. tostring(strError))
		else
			-- Parse the response and return a list
			local back = nil
			local list = {}
			for i = 1, 5 do
			    table.insert(list, { text = "Data " .. i, value = tostring(i) })
			end

			ticket.returnData(list, back)
		end
	end
end

function SelectChannelParamSelect(currentValue, done)
	local ticketId = C4:urlGet("http://www.example.com")
	g_tickets[ticketId] = { returnData = done }
	-- Do not return anything here, we're calling the done function when we have data
end
```

Favoriting Media in MSP Drivers

An end user's ability to designate Navigator UI screens as "Favorites" is a feature introduced in Operating System OS 3. In conjunction with this, drivers written using the Media Service Proxy can provide the ability for end users to designate favorite media delivered through an MSP driver such as Playlists, Albums, Tracks, Stations, Movies, TV Shows, etc.

Inherently, the MSP has no native concept of an album or track. Therefore, a Favoriting Proxy Command and Notification have been added to the OS 3 version of the Proxy. Used together in a MSP driver, these functions support the ability to designate a "Media Favorite" programmatically based on an end user action.

Prior to implementing the new command and notification, a "Favorite to Room" Action should be created by the driver for items that are "pinnable." Pinnable items should include Playlists, Albums, Tracks, Stations, Movies, TV Shows or any other media that could be favorited.

In addition to an action menu icon, a long press should pull up the Action Menu. This will allow drivers to declare actions on link items which do not display the Action Menu icon.

Proxy Command
FavoriteCommand
When a user presses an MSP favorite, Navigator will send the command defined in a new "FavoriteCommand" in the driver's UI capabilities.
Command parameters will reference the Protocol Notification's context. If a driver supports favoriting then a favorite command MUST be defined.

For example:

```
<FavoriteCommand>
    <Name>jumpToFavorite</Name>
    <Type>PROTOCOL</Type>
    <Params>
        <Param>
            <Name>id</Name>
            <Type>FIRST_SELECTED</Type>
            <!-- Value references the favorite id property that was saved from a favorite response
                (FavoriteResponse > Context > favoriteId) -->
            <Value>favoriteId</Value>
        </Param>
   </Params>
</FavoriteCommand>
```

When the FavoriteCommand is fired the driver should return a response instructing Navigator to go to now playing (if playback was started) or go to a particular tab and screen. An empty response can be returned if no action is needed. If the favorite is no longer available the driver should respond with a DATA_RECEIVED proxy notification that

has an ERROR parameter. For example if a favorited album is no longer available the ERROR parameter should inform the user that it is no longer available.

Protocol Notification
When a user creates a favorite by invoking a "Favorite to Room" action, the driver should send a "FavoriteResponse" as the response to the action. The ImageUrl references are unlimited. Navigators will then create a favorite based on the information in the response.

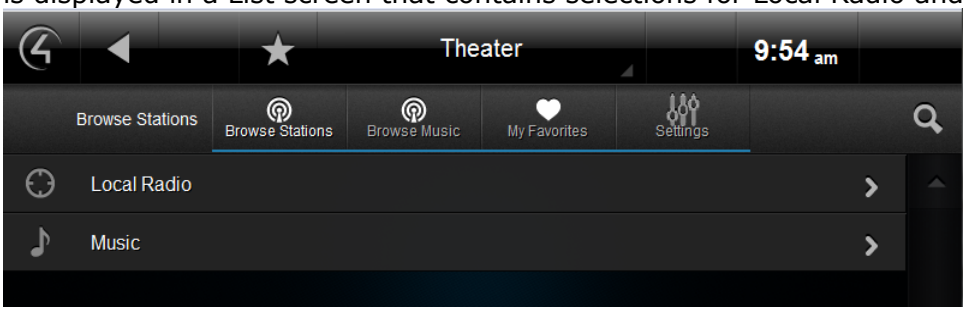
For example:

```
<FavoriteResponse>
    <Title>The favorites title</Title>
    <ImageUrl width="50" height="50">Url for the image</ImageUrl>
    <ImageUrl width="100" height="100">Url for the image</ImageUrl>
    <Context>
        <!-- XML context similar to a notification context -->
        <favoriteId>id|url|something else</favoriteId>
    </Context>
</FavoriteResponse>
```

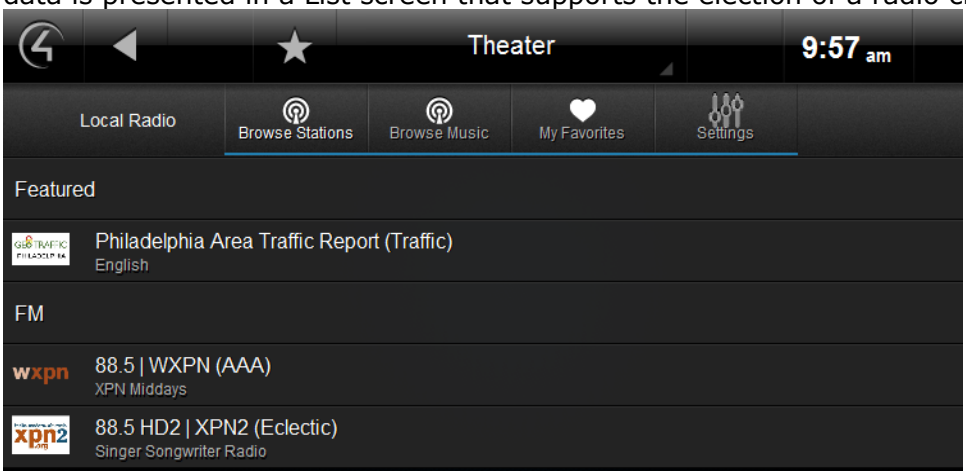
Global Tables Used in the Hello World Driver

The following tables are used in the sample Hello World driver:

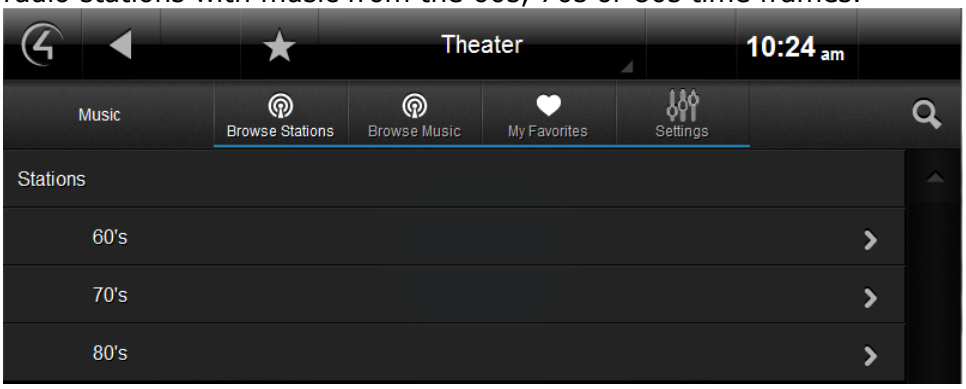
g_browse_mainmenu
This table contains data that populates the main menu for the Hello World driver. This data is displayed in a List screen that contains selections for Local Radio and Music:



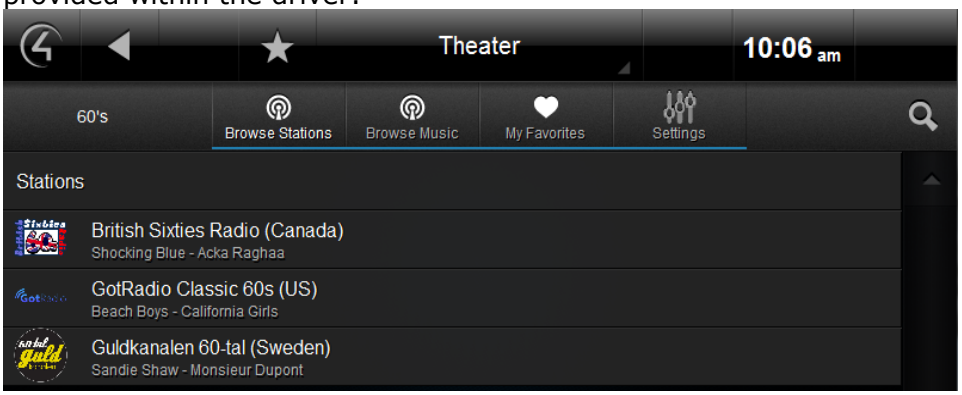
g_browse_localradio
This table contains data that populates the Hello World driver's Local Radion screen. The data is presented in a List screen that supports the election of a radio channel:



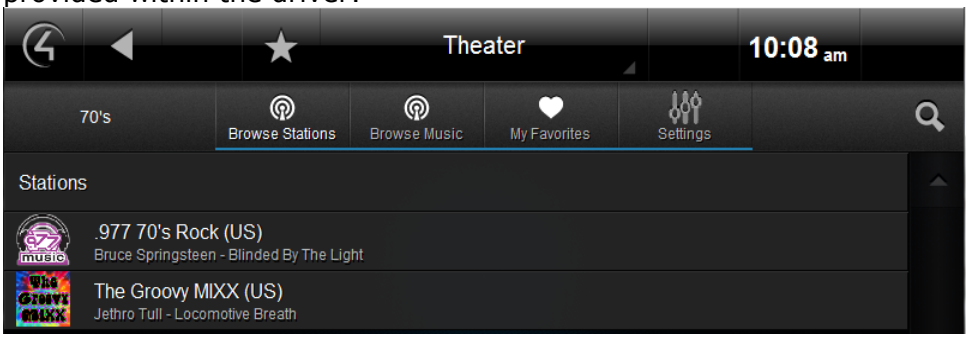
g_browse_stations
This table contains data that populates the Hello World driver's screen when the Music option is selected. The data is presented in a List screen that allows for the selection of radio stations with music from the 60s, 70s or 80s time frames.



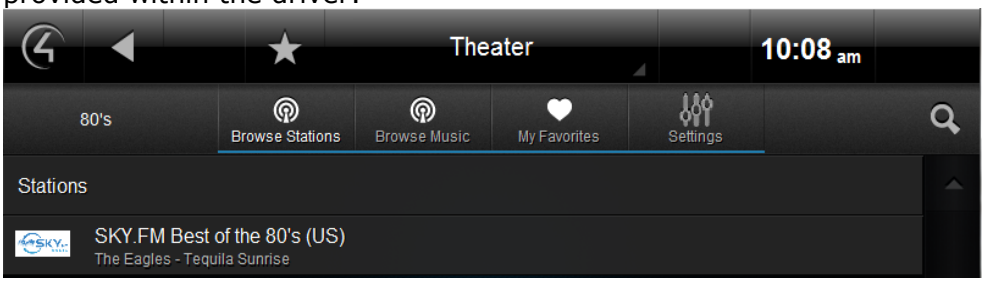
g_browse_stations_60s
This table contains data that populates the Hello World driver's 60's station screen. The data is presented in a Collection screen and supports the selection of 60s themed stations provided within the driver:



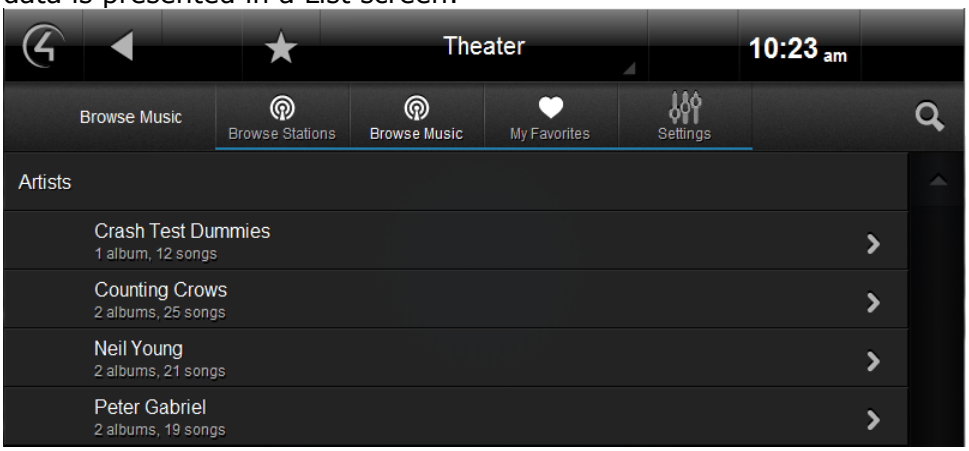
g_browse_stations_70s
This table contains data that populates the Hello World driver's 70's station screen. The data is presented in a Collection screen and supports the selection of a 70s themed stations provided within the driver:



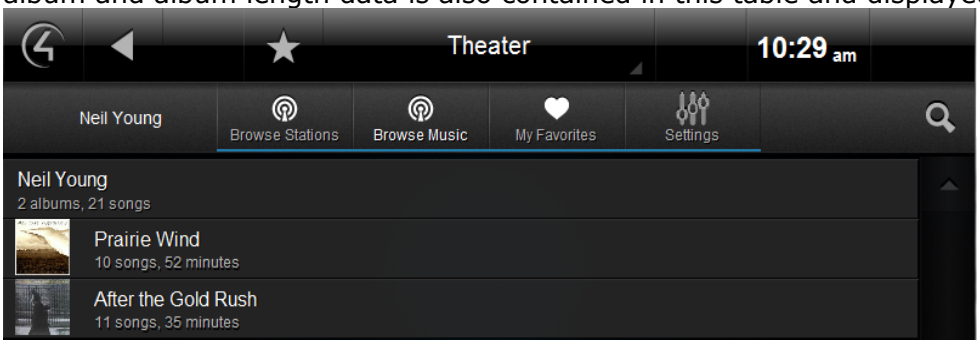
g_browse_stations_80s
This table contains data that populates the Hello World driver's 80's station screen. The data is presented in a Collection screen and supports the selection of a 80s themed stations provided within the driver:



g_browse_music
This table contains data that populates the Hello World driver's screen when the user selects the Browse Music tab from within the Hello World driver. It displays a list of artists along with how many albums and songs are provided from within the Hellow World driver. This data is presented in a List screen:

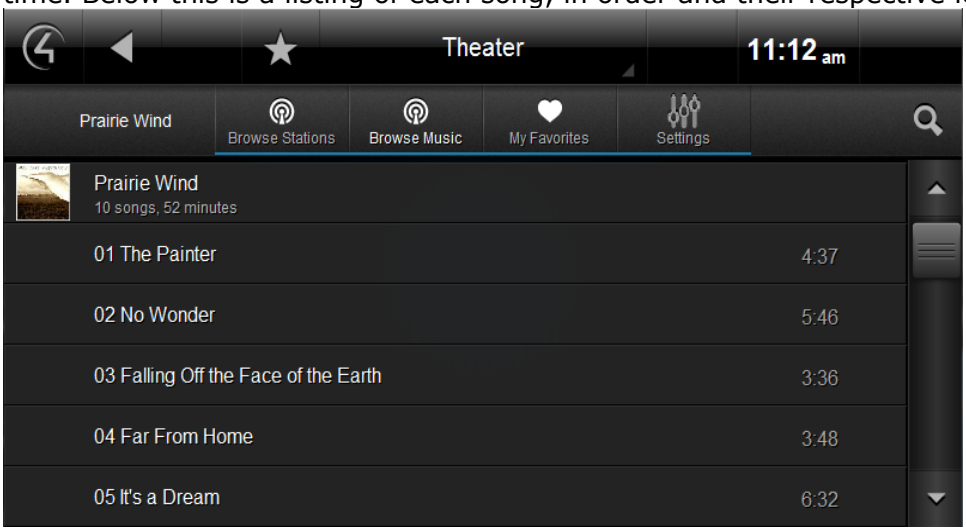


g_music_albums
This table contains data that populates the Hello World driver's album specific data. This data is displayed when a user selects an Artist from the Artist screen. It displays the all of the albums by the selected artist, which are contained in the Hello World driver. Songs per album and album length data is also contained in this table and displayed on the screen:



g_music_SongsByAlbum

This table contains data that populates the Hello World driver's screen when a user selects a specific album. The table is arranged in groups of songs, by album. The data is displayed on the screen with an Album header that displays the number of songs followed by total plying time. Below this is a listing of each song, in order and their respective length:



Glossary

Actions - Actions are similar to device specific commands. However, they are primarily intended to be used during driver installation and configuration. They can only be activated on the Actions tab in Composer and are not available for programming within the system.

BuildListXml - A helper function in the Hello World driver. It creates a table of parameters that consist of data formatted in correct XML.

Capabilities - Capabilities are defined in a proxy and referenced in protocol drivers. They represent typical functionality and physical characteristics found in devices of a given device class. Values for each Capability are entered in a protocol driver and then in-turn, exposed to the Proxy. This enables the proxy driver to determine what capabilities are supported by the device defined by the protocol.

Connections - Connections are the bindings defined in a driver and used within the Control4 system. Generally speaking, they are visible through Composer. In the case of AV, Control and Room bindings – they are reported to the proxy driver when they are bound together in a project.

Connection Class - Connection class information is defined inside each connection instance – within the connections code block of a driver. A connection class lists the physical data or signal type that a specific connection can to transmit.

Command - A command comes from the Control4 system and its destination is a device or a driver.

Data Command - When a screen loads, the first thing it does is fire a DataCommand. The purpose of the DataCommand is to provide the protocol driver a definition to use to build the initial list of data which is being requested from the UI.

Default Action - The Default Action is fired when a user selects an item from the screen.

Director - Director is the software platform that runs on all Control4 controllers. It communicates with device drivers in a manner that enforces a standardized API which hides the complexity of a huge array of different systems and protocols. It also uses that communication to serve up various user interfaces that are displayed on a wide variety of navigation devices.

Driver – A term used to describe files used in the Control Operating System. They are text based files that have an extension of c4i. Several types of drivers exist including *proxy drivers*, *protocol drivers*, *combo drivers* and *multi-proxy drivers*.

Navigator – Refers to an instance of the Control4 UI running on a device.

List Navigator – A text based Navigator instance that is used on Control4 Remote controls

Flash Navigator – A graphical interface that is displayed on touchscreens and monitor-type devices.

Android Navigator – A graphical interface displayed on devices running the Android OS platform
iOS Navigator - A graphical interface displayed on devices running the iOS platform.

Navigator Device – Refers to any device that is running a version of the Control4 Navigator. This includes TVs (onscreen navigator), touchscreens, remote controls, etc.

Notification – Notifications are updates sent to the Control4 system that communicate status or update changes. These notifications may come from a device or the driver.

ParseProxyCommandArgs – A helper function in the Hello World driver. It creates a table of parameters used to assist with a command coming down from the proxy in XML format and modifying that data into a table format which is much more suitable for Lua.

Proxy Binding ID
A proxy binding id is a numerical value that is assigned to each proxy used within a protocol driver. It is a unique reference that can be utilized in the protocol driver code. . This id value is used often when sending data and ensures that the correct proxy-data relationship is always enforced.

Properties
DriverWorks Properties, as defined in the .c4i file, are exposed in the Composer System Design interface on the Properties tab. Properties are data values which are used within your driver. These are initialized in the driver code and also may be configured through the Composer Properties page for your driver. All properties can have initial/default values and may be changed by the installer or configured as read-only. Properties persist automatically so it is not necessary to manually persist them.

Protocol – Protocols contain the definition for a specific device. They are comprised of functionality that is unique to a specific device.

Protocol Driver – An implementation of the protocol in the form of a driver (.c4i) file. When this driver is added to a Control4 Automation system, device control is dictated by the functions defined in the protocol by the device manufacturer. User interaction with the device is provided from a proxy or proxies in the form of the Control4 Navigator UI. User interaction is also supported with system programming such as button presses and events.

Proxy – A Proxy contains the definition for a specific class of device. Proxies are comprised of the most common examples of functionality for each device class. The use of a proxy provides a common interface between the Control4 Navigator UI and a device driver. Proxies are pre-defined and supplied by Control4.

Proxy Driver – An implementation of a proxy in the form of a driver (.c4i) file. When this file is included in a Control4 Automation system, device control includes that of the pre-defined functionality provided in the proxy. User interaction is provided through a Control4 Navigator interface.

UI – User Interface. General term referring to the manner in which a user interfaces with the Control4 system.

Navigator UI – A user interface that is graphically displayed on devices or may also refer to a text based user interface as in List Navigator on remote controls.

Programming UI – A user interface that does not utilize a graphical or text based user interface. This is usually implemented through programming, button presses, timers or events.

This page ends the Media Service Proxy Driver Development documentation.