sdl

iOS Guides

Document current as of 02/16/2023 03:28 PM.

Installation

In order to build your app on a SmartDeviceLink (SDL) Core, the SDL software development kit (SDK) must be installed in your app. The following steps will guide you through adding the SDL SDK to your workspace and configuring the environment.

> **NOTE**
>
> The SDL SDK is currently supported on iOS 10.0 and above.

Install SDL SDK

There are four different ways to install the SDL SDK in your project: Carthage, CocoaPods, Swift Package Manager, or manually.

CocoaPods Installation

1. Xcode should be closed for the following steps.

2. Open the terminal app on your Mac.

3. Make sure you have the latest version of CocoaPods installed. For more information on installing CocoaPods on your system please consult: https://cocoapods.org.

```
sudo gem install cocoapods
```

4. Navigate to the root directory of your app. Make sure your current folder contains the **.xcodeproj** file.

5. Create a new **Podfile**.

```
pod init
```

6. In the **Podfile**, add the following text. This tells CocoaPods to install SDL SDK for iOS. SDL Versions are available on Github. We suggest always using the latest release.

```
target '<#Your Project Name#>' do
    pod 'SmartDeviceLink', '~> <#SDL Version#>'
end
```

7. Install SDL SDK for iOS:

```
pod install
```

8. There will be a newly created **.xcworkspace** file in the directory in addition to the **.xcodeproj** file. Always use the **.xcworkspace** file from now on.

9. Open the **.xcworkspace** file. To open from the terminal, type:

```
open <#Your Project Name#>.xcworkspace
```

Swift Package Manager Installation

You can install this library using the Swift Package Manager. You can install SDL into your iOS project using Xcode 12 by following these steps:

1. Open File -> Swift Packages -> Add Package Dependency...

2. Enter the URL `https://github.com/smartdevicelink/sdl_ios.git` into the search box.

3. Use the default rules or customize the rules to use a specific version or branch. This library added SPM support in version 7.0.0, so please use at least that version.

4. You will be asked which package project to use. If you are using a Swift project, then you should use the `SmartDeviceLinkSwift` project. If not, then you should use the `SmartDeviceLink` project. You can use the `SmartDeviceLink` project in a Swift project as well, but you will miss some Swift specific customizations, which are currently limited to logging enhancements.

5. In your SDL related code, use `import SmartDeviceLink` to call most SDL-related code. If you want to use the Swift-specific logging enhancements you must also use `import SmartDeviceLinkSwift`.

Carthage Installation

SDL iOS supports Carthage! Install using Carthage by following this guide.

Manual Installation

Tagged to our releases is a dynamic framework file that can be drag-and-dropped into the application.

> **NOTE**
>
> You cannot submit your app to the app store with the framework as is. You MUST strip the simulator part of the framework first.

You can check the architectures of your built framework like so:

```
lipo -info SmartDeviceLink.framework/SmartDeviceLink
```

Use a script like this to strip the simulator part of the framework.

```
lipo -remove i386 -remove x86_64 -o SmartDeviceLink.framework/SmartDeviceLink
SmartDeviceLink.framework/SmartDeviceLink
```

SDK Configuration

1. Get an App Id

An app id is required for production level apps. The app id gives your app special permissions to access vehicle data. If your app does not need to access vehicle data, a dummy app id (i.e. creating a fake id like "1234") is sufficient during the development stage. However, you must get an app id before releasing the app to the public.

To obtain an app id, sign up at smartdevicelink.com.

2. Enable Background Capabilities

Your application must be able to maintain a connection to the SDL Core even when it is in the background. This capability must be explicitly enabled for your application (available for iOS 5+). To enable the feature, select your application's build target, go to *Capabilities*, *Background Modes*, and select *External accessory communication mode*.

3. Add SDL Protocol Strings

Your application must support a set of SDL protocol strings in order to be connected to SDL enabled head units. Go to your application's **.plist** file and add the following code under the top level dictionary.

NOTE

This is only required for USB and Bluetooth enabled head units. It is not necessary during development using SDL Core.

```
<key>UISupportedExternalAccessoryProtocols</key>
<array>
<string>com.smartdevicelink.prot29</string>
<string>com.smartdevicelink.prot28</string>
<string>com.smartdevicelink.prot27</string>
<string>com.smartdevicelink.prot26</string>
<string>com.smartdevicelink.prot25</string>
<string>com.smartdevicelink.prot24</string>
<string>com.smartdevicelink.prot23</string>
<string>com.smartdevicelink.prot22</string>
<string>com.smartdevicelink.prot21</string>
<string>com.smartdevicelink.prot20</string>
<string>com.smartdevicelink.prot19</string>
<string>com.smartdevicelink.prot18</string>
<string>com.smartdevicelink.prot17</string>
<string>com.smartdevicelink.prot16</string>
<string>com.smartdevicelink.prot15</string>
<string>com.smartdevicelink.prot14</string>
<string>com.smartdevicelink.prot13</string>
<string>com.smartdevicelink.prot12</string>
<string>com.smartdevicelink.prot11</string>
<string>com.smartdevicelink.prot10</string>
<string>com.smartdevicelink.prot9</string>
<string>com.smartdevicelink.prot8</string>
<string>com.smartdevicelink.prot7</string>
<string>com.smartdevicelink.prot6</string>
<string>com.smartdevicelink.prot5</string>
<string>com.smartdevicelink.prot4</string>
<string>com.smartdevicelink.prot3</string>
<string>com.smartdevicelink.prot2</string>
<string>com.smartdevicelink.prot1</string>
<string>com.smartdevicelink.prot0</string>
<string>com.smartdevicelink.multisession</string>
<string>com.ford.sync.prot0</string>
</array>
```

Integration Basics

Set Up a Proxy Manager Class

You will need a class that manages the connection between your app and SDL Core. Since there should be only one active connection to the SDL Core, you may wish to implement this proxy class using the singleton pattern.

OBJC | SWIFT

Your app should always start passively watching for a connection with a SDL Core as soon as the app launches. The easy way to do this is by instantiating the `ProxyManager` class in the `didFinishLaunchingWithOptions()` method in your `AppDelegate` class.

The connect method will be implemented later. To see a full example, navigate to the bottom of this page.

OBJC | SWIFT

Importing the SDL Library

At the top of the `ProxyManager` class, import the SDL for iOS library.

OBJC | SWIFT

Creating the SDL Manager

The `SDLManager` is the main class of SmartDeviceLink. It will handle setting up the initial connection with the head unit. It will also help you upload images and send RPCs.

OBJC | SWIFT

1. Create a Lifecycle Configuration

In order to instantiate the `SDLManager` class, you must first configure an `SDLConfiguration`. To start, we will look at the `SDLLifecycleConfiguration`. You will at minimum need a `SDLLifecycleConfiguration` instance with the application name and application id.

During the development stage, a dummy app id is usually sufficient. For more information about obtaining an application id, please consult the SDK Configuration section of this guide. You must also decide which network configuration to use to connect the app to the SDL Core. Optional, but recommended, configuration properties include short app name, app icon, and app type.

NETWORK CONNECTION TYPE

There are two different ways to connect your app to a SDL Core: with a TCP (Wi-Fi) network connection or with an iAP (USB / Bluetooth) network connection. Use TCP for debugging and use iAP for production level apps.

IAP

OBJC | SWIFT

TCP

OBJC | SWIFT

> **NOTE**
>
> If you are connecting your app to an emulator using a TCP connection, the IP address is your computer or virtual machine's IP address, and the port number is usually 12345. If you are connecting to Manticore, the Manticore UI will give you your IP / Port to connect to.

2. Short App Name (optional)

This is a shortened version of your app name that is substituted when the full app name will not be visible due to character count constraints. You will want to make this as short

as possible.

OBJC | SWIFT

3. App Icon

This is a custom icon for your application. Please refer to Adaptive Interface Capabilities for icon sizes.

OBJC | SWIFT

> **NOTE**
>
> Persistent files are used when the image ought to remain on the remote system between ignition cycles. This is commonly used for menu artwork, soft button artwork and app icons. Non-persistent artwork is usually used for dynamic images like music album artwork.

4. App Type (optional)

The app type is used by car manufacturers to decide how to categorize your app. Each car manufacturer has a different categorization system. For example, if you set your app type as media, your app will also show up in the audio tab as well as the apps tab of Ford's SYNC® 3 head unit. The app type options are: default, communication, media (i.e. music/podcasts/radio), messaging, navigation, projection, information, and social.

> **NOTE**
>
> Navigation and projection applications both use video and audio byte streaming. However, navigation apps require special permissions from OEMs, and projection apps are only for internal use by OEMs.

OBJC | SWIFT

ADDITIONAL APP TYPES

If one app type doesn't cover your full app use-case, you can add additional `AppHMIType`s as well.

OBJC | SWIFT

5. Template Coloring

You can customize the color scheme of your templates. For more information, see the Customizing the Template guide section.

6. Configure Module Support

You have the ability to determine a minimum SDL protocol and minimum SDL RPC version that your app supports. You can also check the connected vehicle type and disconnect if the vehicle module is not supported. We recommend not setting these values until your app is ready for production. The OEMs you support will help you configure correct values during the application review process.

BLOCKING BY VERSION

If a head unit is blocked by protocol version, your app icon will never appear on the head unit's screen. If you configure your app to block by RPC version, it will appear and then quickly disappear. So while blocking with `minimumProtocolVersion` is preferable, `minimumRPCVersion` allows you more granular control over which RPCs will be present.

OBJC | SWIFT

BLOCKING BY VEHICLE TYPE

If you are blocking by vehicle type and you are connected over RPC v7.1+, your app icon will never appear on the head unit's screen. If you are connected over RPC v7.0 or below, it will appear and then quickly disappear. To implement this type of blocking, you need to implement the SDLManager delegate. You will then implement the optional `didReceiveSystemInfo` method and return `YES` if you want to continue the connection and `NO` if you wish to disconnect. See the section example implementation of a proxy class for an example.

7. Lock Screen

A lock screen is used to prevent the user from interacting with the app on the smartphone while they are driving. When the vehicle starts moving, the lock screen is activated. Similarly, when the vehicle stops moving, the lock screen is removed. You must implement a lock screen in your app for safety reasons. Any application without a lock screen will not get approval for release to the public.

The SDL SDK can take care of the lock screen implementation for you, automatically using your app logo and the connected vehicle logo. If you do not want to use the default lock screen, you can implement your own custom lock screen.

For more information, please refer to the Adding the Lock Screen section; for this guide we will be using `SDLLockScreenConfiguration`'s basic `enabledConfiguration`.

OBJC | SWIFT

8. Logging

A logging configuration is used to define where and how often SDL will log. It will also allow you to set your own logging modules and filters. For more information about setting up logging, see the logging guide.

OBJC | SWIFT

9. File Manager

The file manager configuration allows you to configure retry behavior for uploading files and images. The default configuration attempts one re-upload, but will fail after that.

OBJC | SWIFT

10. Set the Configuration

The `SDLConfiguration` class is used to set the lifecycle, lock screen, logging, and optionally (dependent on if you are a Navigation or Projection app) streaming media configurations for the app. Use the lifecycle configuration settings above to instantiate a `SDLConfiguration` instance.

OBJC | SWIFT

11. Create a SDLManager

Now you can use the `SDLConfiguration` instance to instantiate the `SDLManager`.

OBJC | SWIFT

12. Start the SDLManager

The manager should be started as soon as possible in your application's lifecycle. We suggest doing this in the `didFinishLaunchingWithOptions()` method in your `AppDelegate` class. Once the manager has been initialized, it will immediately start watching for a connection with the remote system. The manager will passively search for a connection with a SDL Core during the entire lifespan of the app. If the manager detects a connection with a SDL Core, the `startWithReadyHandler` will be called.

Create a new function in the `ProxyManager` class called `connect`.

OBJC | SWIFT

> **NOTE**
>
> In production, your app will be watching for connections using iAP, which will not use any more battery power than normal.

If the connection is successful, you can start sending RPCs to the SDL Core. However, some RPCs can only be sent when the HMI is in the `FULL` or `LIMITED` state. If the SDL Core's HMI is not ready to accept these RPCs, your requests will be ignored. If you want to make sure that the SDL Core will not ignore your RPCs, use the `SDLManagerDelegate` methods in the next section.

IMPLEMENT THE SDL MANAGER DELEGATE

The `ProxyManager` class should conform to the `SDLManagerDelegate` protocol. This means that the `ProxyManager` class must implement the following required methods:

1. `managerDidDisconnect` This function is called when the proxy disconnects from the SDL Core. Do any cleanup you need to do in this function.
2. `hmiLevel:didChangeToLevel:` This function is called when the HMI level changes for the app. The HMI level can be `FULL`, `LIMITED`, `BACKGROUND`, or `NONE`. It is important to note that most RPCs sent while the HMI is in `BACKGROUND` or `NONE` mode will be ignored by the SDL Core. For more information, please refer to Understanding Permissions.

In addition, there are several optional methods:

1. `audioStreamingState:didChangeToState:` Called when the audio streaming state of this application changes on the remote system. For more information, please refer to Understanding Permissions.

2. `videoStreamingState:didChangeToState:` Called when the video streaming state of this application changes on the remote system. For more information, please refer to Understanding Permissions.
3. `systemContext:didChangeToContext:` Called when the system context (i.e. a menu is open, an alert is visible, a voice recognition session is in progress) of this application changes on the remote system. For more information, please refer to Understanding Permissions.
4. `managerShouldUpdateLifecycleToLanguage:hmiLanguage:` Called when the module's HMI language or voice recognition language does not match the `language` set in the `SDLLifecycleConfiguration` but does match a language included in `languagesSupported`. If desired, you can customize the `appName`, the `shortAppName`, and `ttsName` for the head unit's current language. For more information about supporting more than one language in your app please refer to Getting Started/Adapting to the Head Unit Language.
5. `didReceiveSystemInfo` Called when the module receives vehicle information, which is before RPC connection on RPC v7.1+ and after RPC connection on RPC v7.0 or below. Returning `YES` will continue the connection, and returning `NO` will cause your app to disconnect from the module.

Example Implementation of a Proxy Class

The following code snippet has an example of setting up both a TCP and iAP connection.

OBJC | SWIFT

Where to Go From Here

You should now be able to connect to a head unit or emulator. For more guidance on connecting, see Connecting to an Infotainment System. To start building your app, learn about designing your interface. Please also review the best practices for building an SDL app.

Connecting to an Infotainment System

In order to view your SDL app, you must connect your device to a head unit that supports SDL Core. If you do not have access to a head unit, we recommend using the Manticore web-based emulator for testing how your SDL app reacts to real-world vehicle events, on-screen interactions and voice recognition.

You will have to configure different connection types based on whether you are connecting to a head unit or an emulator. When connecting to a head unit, you must configure an `iAP` connection. Likewise, when connecting to an emulator, a `TCP` connection must be configured.

Connecting to an Emulator

To connect to an emulator such as Manticore or a local Ubuntu SDL Core-based emulator you must implement a TCP connection when configuring your SDL app.

Getting the IP Address and Port

GENERIC SDL CORE

To connect to a virtual machine running the Ubuntu SDL Core-based emulator, you will use the IP address of the Ubuntu OS and `12345` for the port. You may have to enable port forwarding on your virtual machine if you want to connect using a real device instead of a simulated device.

MANTICORE

Once you launch an instance of Manticore, you will be given an IP address and port number that you can use to configure your TCP connection.

Setting the IP Address and Port

OBJC | SWIFT

Connecting to a Head Unit

To connect your device directly to a production vehicle head unit or Test Development Kit (TDK), make sure to implement an `iAP` connection. Then connect the device using a USB cord or, if the head unit supports it, Bluetooth.

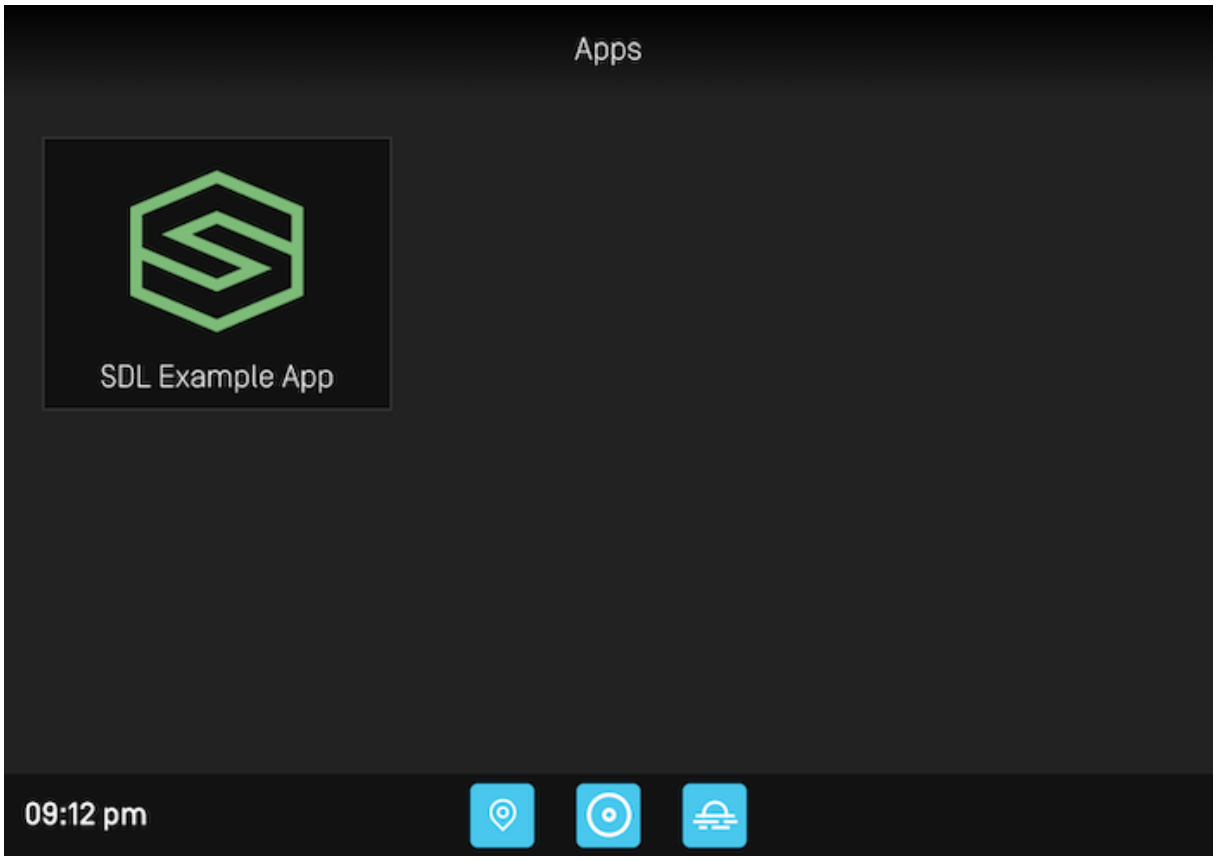
OBJC | SWIFT

Viewing Realtime Logs

If you are testing with a vehicle head unit or TDK and wish to see realtime debug logs in the Xcode console, you should use wireless debugging.

Running the SDL App

Build and run the project in Xcode, targeting the device or simulator that you want to test your app with. Your app should compile and launch on your device of choosing. If your connection configuration is setup correctly, you should see your SDL app icon appear on the HMI screen:



To open your app, click on your app's icon in the HMI.

This is the main screen of your SDL app. If you get to this point, your SDL app is working.

Troubleshooting

If you are having issues with connecting to an emulator or head unit, please see our troubleshooting tips in the Example Apps section of the guide.

Adding the Lock Screen

The lock screen is a vital part of your SDL app because it prevents the user from using the phone while the vehicle is in motion. SDL takes care of the lock screen for you. If you prefer your own look, but still want the recommended logic that SDL provides for free, you can also set your own custom lock screen.

If you would not like to use any of the following code, you may use the `SDLLockScreenConfiguration` class function `disabledConfiguration`, and manage the entire lifecycle of the lock screen yourself. However, it is strongly recommended that you use the provided lock screen manager, even if you use your own view controller.

To see where the `SDLLockScreenConfiguration` is used, refer to the Integration Basics guide.

Using the Provided Lock Screen

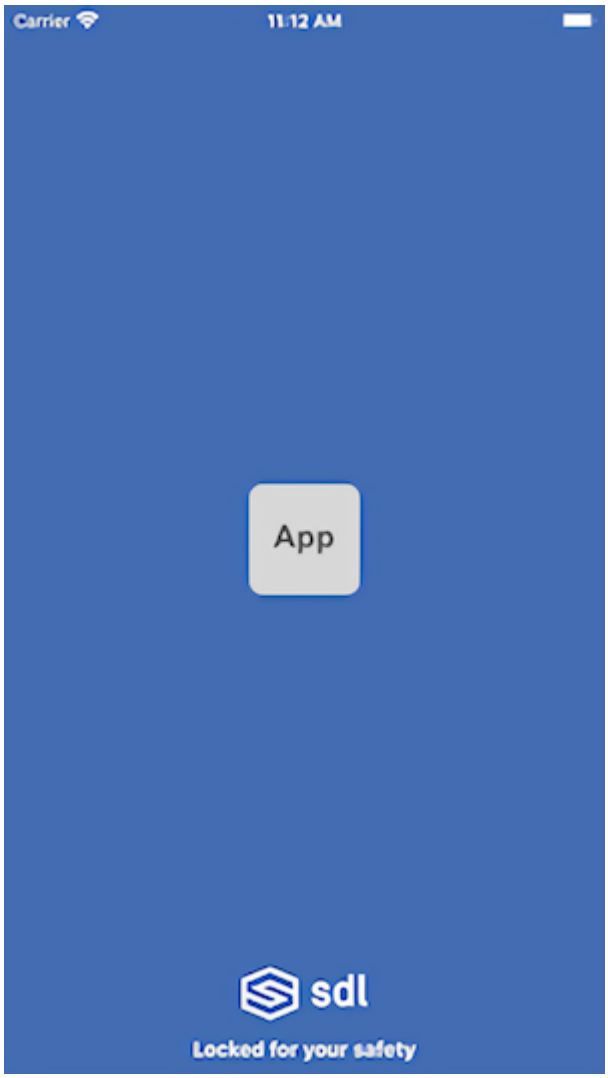
Using the default lock screen is simple. Using the lock screen this way will automatically load an automaker's logo, if available, to show alongside your logo. If it is not, the default lock screen will show your logo alone.

To do this, instantiate a new `SDLLockScreenConfiguration`:

OBJC | SWIFT

Customizing the Default Lock Screen

It is possible to customize the background color and app icon in the default provided lockscreen. If you choose not to set your own app icon the library will use the SDL logo.



Custom Background Color

OBJC | SWIFT

Custom App Icon

OBJC | SWIFT

Showing the OEM Logo

The default lock screen handles retrieving and setting the OEM logo from head units that support this feature.

This feature can be disabled on the default lock screen by setting `showDeviceLogo` to false.

OBJC | SWIFT

Creating a Custom Lock Screen

If you would like to use your own lock screen instead of the one provided by the library, but still use the logic we provide, you can use a new initializer within `SDLLockScreenConfiguration`. Any custom lock screen you create should be a subclass of `SDLLockScreenView`

`Controller` to ensure that it is configured correctly and can receive all of the information necessary to customize your lock screen such as the OEM icon.

> **NOTE**
>
> If you create a custom lock screen view controller, please note that the view controller's default `view` background will be transparent, even if you set a background color for it. You **must** place a custom view across the entire view controller in order to make your lock screen opaque.

OBJC | SWIFT

Customizing the Lock Screen State

In SDL iOS v6.4, a new parameter `displayMode` has been added to the `SDLLockScreenConfiguration` to control the state of the lock screen and the older boolean parameters have been deprecated.

DISPLAYMODE	DESCRIPTION
never	The lock screen should never be shown. This should almost always mean that you will build your own lock screen
requiredOnly	The lock screen should only be shown when it is required by the head unit
optionalOrRequired	The lock screen should be shown when required by the head unit or when the head unit says that its optional, but *not* in other cases, such as before the user has interacted with your app on the head unit
always	The lock screen should always be shown after connection

Disabling the Lock Screen

Please note that a lock screen will be required by most OEMs. You can disable the lock screen manager, but you will then be required to implement your own logic for showing and hiding the lock screen. This is not recommended as the `SDLLockScreenConfiguration` adheres to most OEM lock screen requirements. However, if you must create a lock screen manager from scratch, the library's lock screen manager can be disabled via the `SDLLockScreenConfiguration` as follows:

OBJC | SWIFT

Making the Lock Screen Always On

The lock screen manager is configured to dismiss the lock screen when it is safe to do so. To always have the lock screen visible when the device is connected to the head unit, simply update the lock screen configuration.

OBJC | SWIFT

Enabling User Lockscreen Dismissal (Passenger Mode)

Starting in RPC v6.0+ users may now have the ability to dismiss the lock screen by swiping the lock screen down. Not all OEMs support this new feature. A dismissible lock screen is enabled by default if the head unit enables the feature, but you can disable it manually as well.

To disable this feature, set `SDLLockScreenConfiguration`s `enableDismissGesture` to false.

OBJC | SWIFT

Multiple Transports (Protocol v5.1+)

The multiple transports feature allows apps to carry their SDL session over multiple transports. The first transport that the app connects with is referred to as the primary transport and a transport connected at a later point is the secondary transport. For example, apps can register over Bluetooth or USB as a primary transport, then connect over WiFi when necessary (ex. to allow video/audio streaming) as a secondary transport. This feature is supported on connections with protocol version 5.1+, which is supported on SDL iOS 6.1+ and SDL Core 5.0+.

Primary Transports

On head units that support multiple transports, the primary transport will be used for RPC communication while the secondary transport will be used for high bandwidth services such as streaming video data for navigation applications. If no high-bandwidth secondary transport is present, the primary transport will be used for all needed services that the transport supports.

The only primary transport available for iOS in production applications is iAP.

Secondary Transports

Secondary transports must be enabled by the module to which the app is connecting. TCP over WiFi can be configured as a supported secondary transport.

By default, TCP is a configured secondary transport, but this can be disabled.

OBJC | SWIFT

Adapting to the Head Unit Language

Since a head unit can support multiple languages, you may want to add support for more than one language to your SDL app. The SDL library allows you to check which language is currently used by the head unit. If desired, the app's name and the app's text-to-speech (TTS) name can be customized to reflect the head unit's current language. If your app name is not part of the current lexicon, you should tell the VR system how a native speaker will pronounce your app name by setting the TTS name using phonemes from either the Microsoft SAPI phoneme set or from the LHPLUS phoneme set.

Setting the Default Language

The initial configuration of the `SDLManager` requires a default language when setting the `SDLLifecycleConfiguration`. If not set, the SDL library uses American English (*EN_US*) as the default language. The connection will fail if the head unit does not support the `language` set in the `SDLLifecycleConfiguration`. The `RegisterAppInterface` response RPC will return `INVALID_DATA` as the reason for rejecting the request.

What if My App Does Not Support the Head Unit Language?

If your app does not support the current head unit language, you should decide on a default language to use in your app. All text should be created using this default language. Unfortunately, your VR commands will probably not work as the VR system will not recognize your users' pronunciation.

Checking the Current Head Unit Language

After starting the `SDLManager` you can check the `registerResponse` property for the head unit's `language` and `hmiDisplayLanguage`. The `language` property gives you the current VR system language; `hmiDisplayLanguage` the current display text language.

OBJC | SWIFT

Updating the SDL App Name

To customize the app name for the head unit's current language, implement the following steps:

1. Set the default `language` in the `SDLLifecycleConfiguration`.
2. Add all languages your app supports to `languagesSupported` in the `SDLLifecycleConfiguration`.
3. Implement the `SDLManagerDelegate`'s `managerShouldUpdateLifecycleToLanguage:hmiLanguage:` method. If the module's current HMI language or voice recognition (VR) language is different from the app's default language, the method will be called with the module's current HMI and/or VR language. Please note that the delegate method will only be called if your app supports the head unit's current language. Return a `SDLLifecycleConfigurationUpdate` object with the new `appName` and/or `ttsName`.
   method. If the module's current HMI language or voice recognition (VR) language is different from the app's default language, the listener will be called with the module's current HMI and/or VR language. Return a `LifecycleConfigurationUpdate` with the new `appName` and/or `ttsName`.

OBJC | SWIFT

Understanding Permissions

While creating your SDL app, remember that just because your app is connected to a head unit it does not mean that the app has permission to send the RPCs you want. If your app

does not have the required permissions, requests will be rejected. There are three important things to remember in regards to permissions:

1. You may not be able to send a RPC when the SDL app is closed, in the background, or obscured by an alert. Each RPC has a set of `hmiLevel`s during which it can be sent.
2. For some RPCs, like those that access vehicle data or make a phone call, you may need special permissions from the OEM to use. This permission is granted when you submit your app to the OEM for approval. Each OEM decides which RPCs it will restrict access to, so it is up you to check if you are allowed to use the RPC with the head unit.
3. Some head units may not support all RPCs.

HMI Levels

When your app is connected to the head unit you will receive notifications when the SDL app's HMI status changes. Your app can be in one of four different `hmiLevel`s:

HMI LEVEL	WHAT DOES THIS MEAN?
NONE	The user has not yet opened your app, or the app has been killed.
BACKGROUND	The user has opened your app, but is currently in another part of the head unit.
LIMITED	This level only applies to media and navigation apps (i.e. apps with an `appType` of `MEDIA` or `NAVIGATION`). The user has opened your app, but is currently in another part of the head unit. The app can receive button presses from the play, seek, tune, and preset buttons.
FULL	Your app is currently in focus on the screen.

Be careful with sending user interface related RPCs in the `NONE` and `BACKGROUND` levels; some head units may reject RPCs sent in those states. We recommended that you wait until your app's `hmiLevel` enters `FULL` to set up your app's UI.

To get more detailed information about the state of your SDL app check the current system context. The system context will let you know if a menu is open, a VR session is in progress, an alert is showing, or if the main screen is unobstructed. You can find more information about the system context below.

Monitoring the HMI Level

The easiest way to monitor the `hmiLevel` of your SDL app is through a required delegate callback of `SDLManagerDelegate`. The function `hmiLevel:didChangeToLevel:` is called every time your app's `hmiLevel` changes.

OBJC | SWIFT

Permission Manager

The PermissionManager allows developers to easily query whether specific RPCs are allowed or not in the current state of the app. It also allows a listener to be added for RPCs or their parameters so that if there are changes in their permissions, the app will be notified.

Checking Current Permissions of a Single RPC

OBJC | SWIFT

Checking Current Permissions of a Group of RPCs

You can also retrieve the status of a group of RPCs. First, you can retrieve the permission status of the group of RPCs as a whole: whether or not those RPCs are all allowed, all disallowed, or some are allowed and some are disallowed. This will allow you to know, for example, if a feature you need is allowed based on the status of all the RPCs needed for the feature.

OBJC | SWIFT

The previous snippet will give a quick generic status for all permissions together. However, if you want to get a more detailed result about the status of every permission or parameter in the group, you can use the `statusesOfRPCPermissions:` method.

OBJC | SWIFT

Observing Permissions

If desired, you can subscribe to a group of permissions. The subscription's handler will be called when the permissions for the group changes. If you want to be notified when the permission status of any of RPCs in the group change, set the `groupType` to `SDLPermissionGroupTypeAny`. If you only want to be notified when all of the RPCs in the group are allowed, or go from allowed to some/all not allowed, set the `groupType` to `SDLPermissionGroupTypeAllAllowed`.

OBJC | SWIFT

Stopping Observation of Permissions

When you set up the subscription, you will get a unique id back. Use this id to unsubscribe to the permissions at a later date.

OBJC | SWIFT

Additional HMI State Information

If you want more detail about the current state of your SDL app you can monitor the audio playback state as well as get notifications when something blocks the main screen of your app.

Audio Streaming State

The Audio Streaming State informs your app whether or not the driver will be able to hear your app's audio. It will be either `AUDIBLE`, `NOT_AUDIBLE`, or `ATTENUATED`.

You will get these notifications when an alert pops up, when you start recording the in-car audio, when voice recognition is active, when another app takes audio control, when a navigation app is giving directions, etc.

AUDIO STREAMING STATE	WHAT DOES THIS MEAN?
AUDIBLE	Any audio you are playing will be audible to the user
ATTENUATED	Some kind of audio mixing is occurring between what you are playing, if anything, and some system level audio or navigation application audio.
NOT_AUDIBLE	Your streaming audio is not audible. This could occur during a `VRSESSION` System Context.

OBJC | SWIFT

System Context

The System Context informs your app if there is potentially a blocking HMI component while your app is still visible. An example of this would be if your application is open and you display an alert. Your app will receive a system context of `ALERT` while it is presented on the screen, followed by `MAIN` when it is dismissed.

SYSTEM CONTEXT STATE	WHAT DOES THIS MEAN?
MAIN	No user interaction is in progress that could be blocking your app's visibility.
VRSESSION	Voice recognition is currently in progress.
MENU	A menu interaction is currently in-progress.
HMI_OBSCURED	The app's display HMI is being blocked by either a system or other app's overlay (another app's alert, for instance).
ALERT	An alert that you have sent is currently visible.

OBJC | SWIFT

Checking Supported Features

New features are always being added to SDL, however, you or your users may be connecting to modules that do not support the newest features. If your SDL app attempts to use an unsupported feature your request will be ignored by the module.

When you are implementing a feature you should always assume that some modules your users connect to will not support the feature or that the user may have disabled permissions for this feature on their head unit. The best way to deal with unsupported features is to check if the feature is available before attempting to use it and to handle error responses.

Checking the System Capability Manager

The easiest way to check if a feature is supported is to query the library's System Capability Manager. For more details on how get this information, please see the Adaptive Interface Capabilities guide.

Handling RPC Error Responses

When you are trying to use a feature, you can watch for an error response to the RPC request you sent to the module. If the response contains an error, you may be able to check the `result` enum to determine if the feature is disabled. If the response that comes back is of the type `GenericResponse`, the module doesn't understand your request.

OBJC | SWIFT

Checking if a Feature is Supported by Version

When you connect successfully to a head unit, SDL will automatically negotiate the maximum SDL RPC version supported by both the module and your SDL SDK. If the feature you want to support was added in a version less than or equal to the version returned by the head unit, then your head unit may support the feature. Remember that the module may still disable the feature, or the user may still have disabled permissions for the feature in some cases. It's best to check if the feature is supported through the System Capability Manager first, but you may also check the negotiated version to know if the head unit was built before the feature was designed.

Throughout these guides you may see headers that contain text like "RPC 6.0+". That means that if the negotiated version is 6.0 or greater, then SDL supports the feature but the above caveats may still apply.

OBJC | SWIFT

Example Apps

SDL provides two example apps: one written in Objective-C and one in Swift. Both implement the same features.

The example apps are located in the sdl_ios repository. To try them, you can download the repository and run the example app targets, or you can use `pod try SmartDeviceLink` with CocoaPods installed on your Mac.

> **NOTE**
>
> If you download or clone the SDL repository in order to run the example apps, you must first obtain the BSON submodule. You can do so by running `git submodule init` and `git submodule update` in your terminal when in the main directory of the cloned repository.

The example apps implement soft buttons, template text and images, a main menu and submenu, vehicle data, popup menus, voice commands, and capturing in-car audio.

Connecting to an Infotainment System

Emulator

You can use a simulated or a real device to connect the example app to an emulator. To connect the example app to Manticore or another emulator, make sure you are on the `TCP Debug` tab of the example app. Then type in the IP address and port number and press the "Connect" button. The button will turn green when you are connected. Please check the Connecting to an Infotainment System guide for more detailed instructions on how to get the emulator's IP address and port number.

Head Unit

You need a real device to connect the example app to production or debug hardware. After building the running the app, make sure you are on the `iAP` tab of the example app and press "Connect". The button will turn green when you are connected.

If using the Bluetooth (BT) transport, make sure to first pair your phone to the hardware before attempting to connect your SDL app. If using the USB transport, you will need to connect your phone to the hardware using a USB cord.

If the hardware supports both BT and USB transports, only one transport will be supported at once. If your phone is connected via BT and you then connect the phone to the head unit via a USB cord, the library will close the BT session and open a new session over USB. Likewise, when the USB cord is disconnected, the library will close the USB session and open session over BT.

Troubleshooting

If your app compiles and but does not show up on the HMI, there are a few things you should check:

TCP Debug Transport

1. Make sure the correct IP address and port number is set in the `SDLLifecycleConfiguration`.
2. Make sure the device and the SDL Core emulator are on the same network.
3. If you are running an SDL Core emulator on a virtual machine, and you are using port forwarding to connect your device to the virtual machine, the IP address should be the IP address of your machine hosting the VM, not the IP address of the VM. The port number will be `12345`.
4. Make sure there is no firewall blocking the incoming port `12345` on the machine or VM running the SDL Core emulator. Also make sure your firewall allows that outgoing port.
5. Your SDL app will not work when the device app is in the background, because the OS will terminate background tasks after a short amount of time. This is not an issue with production IAP connections because Apple's External Accessory framework allows your app unlimited background time.

6. If you have a media SDL app, audio will not play on the emulator. Only production IAP connections are currently able to play audio because this happens over the standard Bluetooth / USB system audio channel.
7. You cannot connect to any of our open-source emulators using a USB cord or Bluetooth because Apple's MFi Program is confidential and can not be used in open source projects.

iAP Production Transport

1. Make sure to use the default `SDLLifecycleConfiguration`.
2. Make sure the protocol strings have been added to the app.
3. Make sure you have enabled background capabilities for your app.
4. If the head unit (emulators do not support IAP) does not support Bluetooth, an iAP connection requires a USB cord.

IAP BLUETOOTH PRODUCTION TRANSPORT

1. Bluetooth transport support is automatic when you support the iAP production transport. It cannot be turned on or off separately.
2. Make sure the head unit supports Bluetooth transport for iPhones. Currently, only some head units support Bluetooth.
3. Make sure Bluetooth is turned on - both on the head unit hardware and your iPhone.
4. Ensure your iPhone is properly paired with the head unit.

Additional Examples

For more examples go to the SmartDeviceLink Examples GitHub organization. Download or clone any of these projects.

The examples available include an example weather app and an example navigation app.

The example weather app uses the OpenWeather API to implement a basic connected weather app with SDL UI. This example showcases changing screen template items for

certain weather forecasts, displaying hourly and daily weather in popup menus, and showing weather alerts with SDL Alerts.

The example navigation app utilizes the MapBox API to create a basic video streaming map app. The example navigation app can be used as a reference for developers who want to create their own navigation app.

> **NOTE**
>
> Some examples require obtaining API tokens from third parties for data and services. For all of these examples follow the setup instructions as outlined in their **README.md**.

Adaptive Interface Capabilities

Since each car manufacturer has different user interface style guidelines, the number of lines of text, soft and hard buttons, and images supported will vary between different types of head units. The system will send information to your app about its capabilities for various user interface elements. You should use this information to create the user interface of your SDL app.

You can access these properties on the `SDLManager.systemCapabilityManager` instance.

System Capability Manager Properties

PARAMETERS	DESCRIPTION	RPC VERSION
displays	Specifies display related information. The primary display will be the first element within the array. Windows within that display are different places that the app could be displayed (such as the main app window and various widget windows).	RPC v6.0+
hmiZoneCapabilities	Specifies HMI Zones in the vehicle. There may be a HMI available for back seat passengers as well as front seat passengers.	RPC v1.0+
speechCapabilities	Contains information about TTS capabilities on the SDL platform. Platforms may support text, SAPI phonemes, LH PLUS phonemes, pre-recorded speech, and silence.	RPC v1.0+
prerecordedSpeechCapabilities	A list of pre-recorded sounds you can use in your app. Sounds may include a help, initial, listen, positive, or a negative jingle.	RPC v3.0+
vrCapability	The voice-recognition capabilities of the connected SDL platform. The platform may be able to recognize spoken text in the current language.	RPC v1.0+

PARAMETERS	DESCRIPTION	RPC VERSION
audioPassThruCapabilities	Describes the sampling rate, bits per sample, and audio types available.	RPC v2.0+
pcmStreamCapabilities	Describes different audio type configurations for the audio PCM stream service, e.g. {8kHz,8-bit,PCM}.	RPC v4.1+
hmiCapabilities	Returns whether or not the app can support built-in navigation and phone calls.	RPC v3.0+
appServicesCapabilities	Describes the capabilities of app services including what service types are supported and the current state of services.	RPC v5.1+
navigationCapability	Describes the built-in vehicle navigation system's APIs.	RPC v4.5+
phoneCapability	Describes the built-in phone calling capabilities of the IVI system.	RPC v4.5+
videoStreamingCapability	Describes the abilities of the head unit to video stream projection applications.	RPC v4.5+
remoteControlCapability	Describes the abilities of an app to control built-in aspects of the IVI system.	RPC v4.5+
seatLocationCapability	Describes the positioning of each seat in a vehicle	RPC v6.0+

Deprecated Properties

The following properties are deprecated on SDL iOS 6.4 because as of RPC v6.0 they are deprecated. However, these properties will still be filled with information. When connected on RPC <6.0, the information will be exactly the same as what is returned in the `RegisterAppInterfaceResponse` and `SetDisplayLayoutResponse`. However, if connected on RPC >6.0, the information will be converted from the newer-style display information, which means that some information will not be available.

PARAMETERS	DESCRIPTION
displayCapabilities	Information about the HMI display. This includes information about available templates, whether or not graphics are supported, and a list of all text fields and the max number of characters allowed in each text field.
buttonCapabilities	A list of available buttons and whether the buttons support long, short and up-down presses.
softButtonCapabilities	A list of available soft buttons and whether the button support images. Also, information about whether the button supports long, short and up-down presses.
presetBankCapabilities	If returned, the platform supports custom on-screen presets.

Image Specifics

Images may be formatted as PNG, JPEG, or BMP. You can find which image types and resolutions are supported using the system capability manager.

Since the head unit connection is often relatively slow (especially over Bluetooth), you should pay attention to the size of your images to ensure that they are not larger than they need to be. If an image is uploaded that is larger than the supported size, the image will be scaled down by Core.

OBJC | SWIFT

EXAMPLE IMAGE SIZES

Below is a table with example image sizes. Check the `SystemCapabilityManager` for the exact image sizes desired by the system you are connecting to. The connected system should be able to scale down larger sizes, but if the image you are sending is much larger than desired, then performance will be impacted.

IMAGENAME	USED IN RPC	DETAILS	SIZE	TYPE
softButtonImage	Show	Image shown on softbuttons on the base screen	70x70px	png, jpg, bmp
choiceImage	CreateInteractionChoiceSet	Image shown in the manual part of an performInteraction either big (ICON_ONLY) or small (LIST_ONLY)	70x70px	png, jpg, bmp
choiceSecondaryImage	CreateInteractionChoiceSet	Image shown on the right side of an entry in (LIST_ONLY) performInteraction	35x35px	png, jpg, bmp
vrHelpItem	SetGlobalProperties	Image shown during voice interaction	35x35px	png, jpg, bmp
menuIcon	SetGlobalProperties	Image shown on the “More…” button	35x35px	png, jpg, bmp
cmdIcon	AddCommand	Image shown for commands in the "More…" menu	35x35px	png, jpg, bmp

IMAGENAME	USED IN RPC	DETAILS	SIZE	TYPE
appIcon	SetAppIcon	Image shown as Icon in the "Mobile Apps" menu	70x70px	png, jpg, bmp
graphic	Show	Image shown on the base screen as cover art	185x185px	png, jpg, bmp

Querying and Subscribing System Capabilities

Capabilities that can be updated can be queried and subscribed to using the `SDLSystemCapabilityManager`.

Determining Support for System Capabilities

You should check if the head unit supports your desired capability before subscribing to or updating the capability.

OBJC | SWIFT

Manual Querying for System Capabilities

Most head units provide features that your app can use: making and receiving phone calls, an embedded navigation system, video and audio streaming, as well as supporting app services. To pull information about this capability, use the `SDLSystemCapabilityManager` to query the head unit for the desired capability. If a capability is unavailable, the query will return `nil`.

OBJC | SWIFT

Subscribing to System Capabilities (RPC v5.1+)

In addition to getting the current system capabilities, it is also possible to subscribe for updates when the head unit capabilities change. To get these notifications you must register using a `subscribeToCapabilityType:` method.

> **NOTE**
>
> If `supportsSubscriptions == NO`, you can still subscribe to capabilities, however, you must manually poll for new capability updates using `updateCapabilityType:completionHandler:`. All subscriptions will be automatically updated when that method returns a new value.
>
> The `DISPLAYS` type can be subscribed on all SDL versions.

CHECKING IF THE HEAD UNIT SUPPORTS SUBSCRIPTIONS

OBJC | SWIFT

SUBSCRIBE TO A CAPABILITY

OBJC | SWIFT

Main Screen Templates

Each head unit manufacturer supports a set of user interface templates. These templates determine the position and size of the text, images, and buttons on the screen. Once the app has connected successfully with an SDL enabled head unit, a list of supported templates is available on `SDLManager.systemCapabilityManager.defaultMainWindowCapability.templatesAvailable`.

Change the Template

To change a template at any time, use `[SDLScreenManager changeLayout:]`. This guide requires SDL iOS version 7.0. If using an older version, use the `SetDisplayLayout` RPC.

> **NOTE**
>
> When changing the layout, you may get an error or failure if the update is "superseded." This isn't technically a failure, because changing the layout has not yet been attempted. The layout or batched operation was cancelled before it could be completed because another operation was requested. The layout change will then be inserted into the future operation and completed then.

OBJC | SWIFT

Template changes can also be batched with text and graphics updates:

OBJC | SWIFT

When changing screen layouts and template data (for example, to show a weather hourly data screen vs. a daily weather screen), it is recommended to encapsulate these updates into a class or method. Doing so is a good way to keep SDL UI changes organized. A fully-formed example of this can be seen in the example weather app. Below is a generic example.

Screen Change Example Code

This example code creates an interface that can be implemented by various "screens" of your SDL app. This is a recommended design pattern so that you can separate your code to only involve the data models you need. This is just a simple example and your own needs may be different.

Screen Change Example Interface

All screens will need to have access to the `SDLScreenManager` object and a function to display the screen. Therefore, it is recommended to create a generic interface for all screens to follow. For the example below, the `CustomSDLScreen` protocol requires an initializer with the parameters `SDLManager` and a `showScreen` method.

OBJC | SWIFT

Screen Change Example Implementations

The following example code shows a few implementations of the example screen changing protocol. A good practice for screen classes is to keep screen data in a view model. Doing so will add a layer of abstraction for exposing public properties and commands to the screen.

For the example below, the `HomeScreen` class will inherit the `CustomSDLScreen` interface and will have a property of type `HomeDataViewModel`. The screen manager will change its text fields based on the view model's data. In addition, the home screen will also create a navigation button to open the `ButtonSDLScreen` when pressed.

OBJC | SWIFT

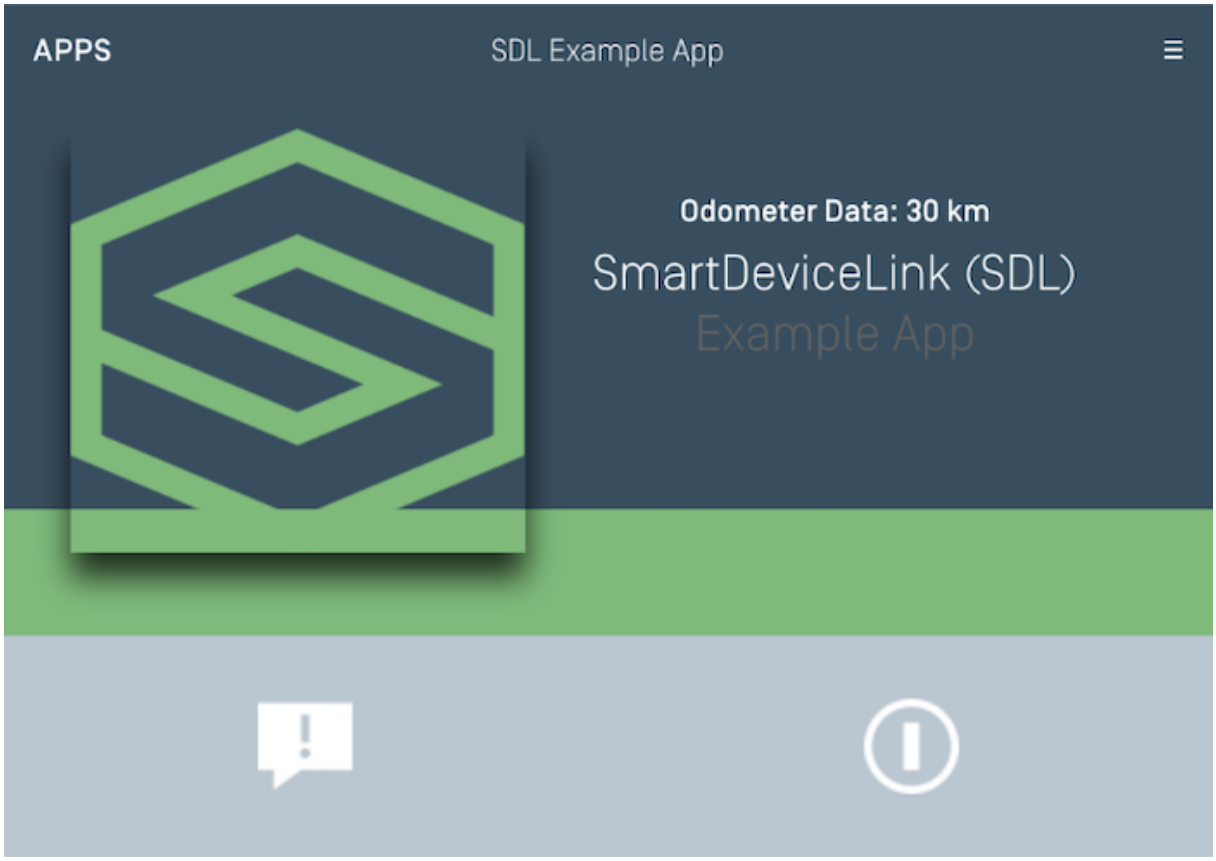
The `ButtonSDLScreen` follows the same patterns as the `HomeSDLScreen` but has minor implementation differences. The screen's view model `ButtonDataViewModel` contains properties unique to the `ButtonSDLScreen` such as text fields and an array of soft button objects. It also changes the template configuration to tiles only.

OBJC | SWIFT

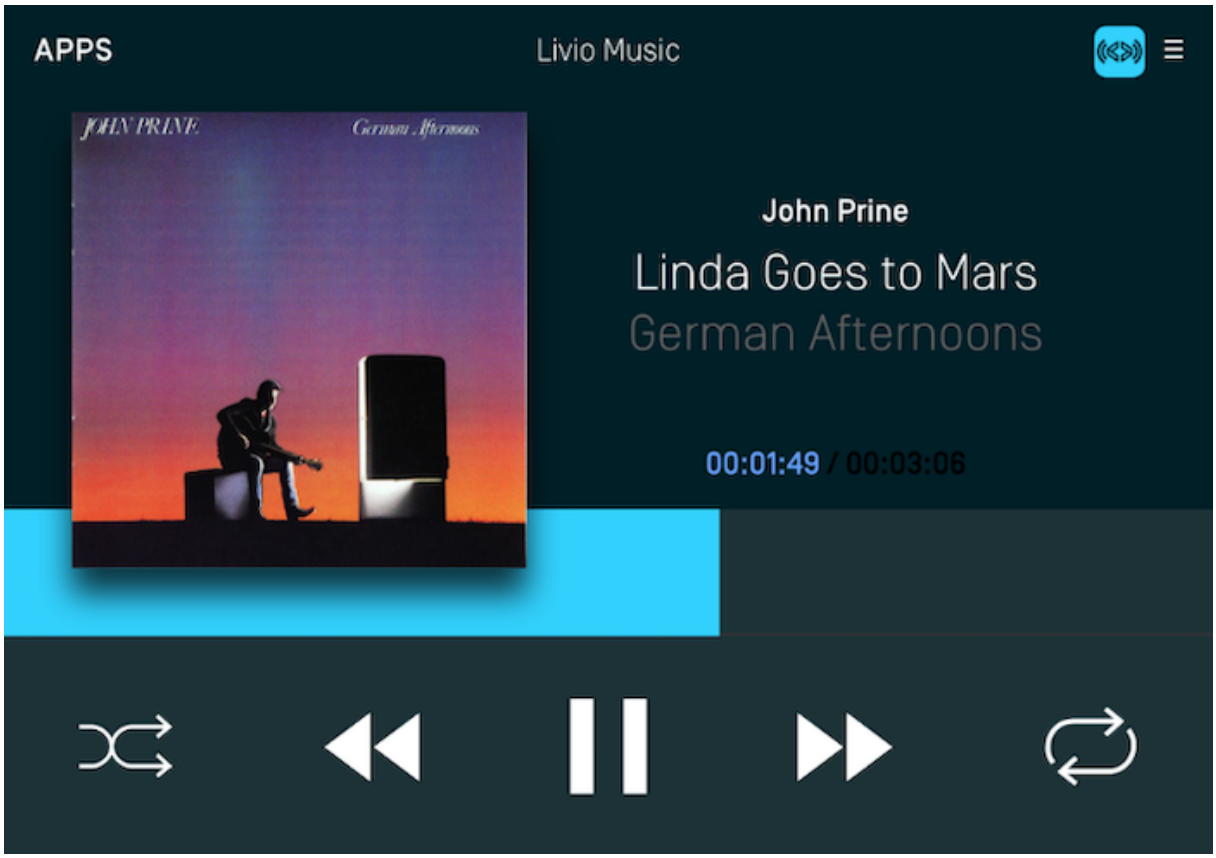
Available Templates

There are fifteen standard templates to choose from, however some head units may only support a subset of these templates. The following examples show how templates will appear on the Generic HMI and Ford's SYNC® 3 HMI.

MEDIA



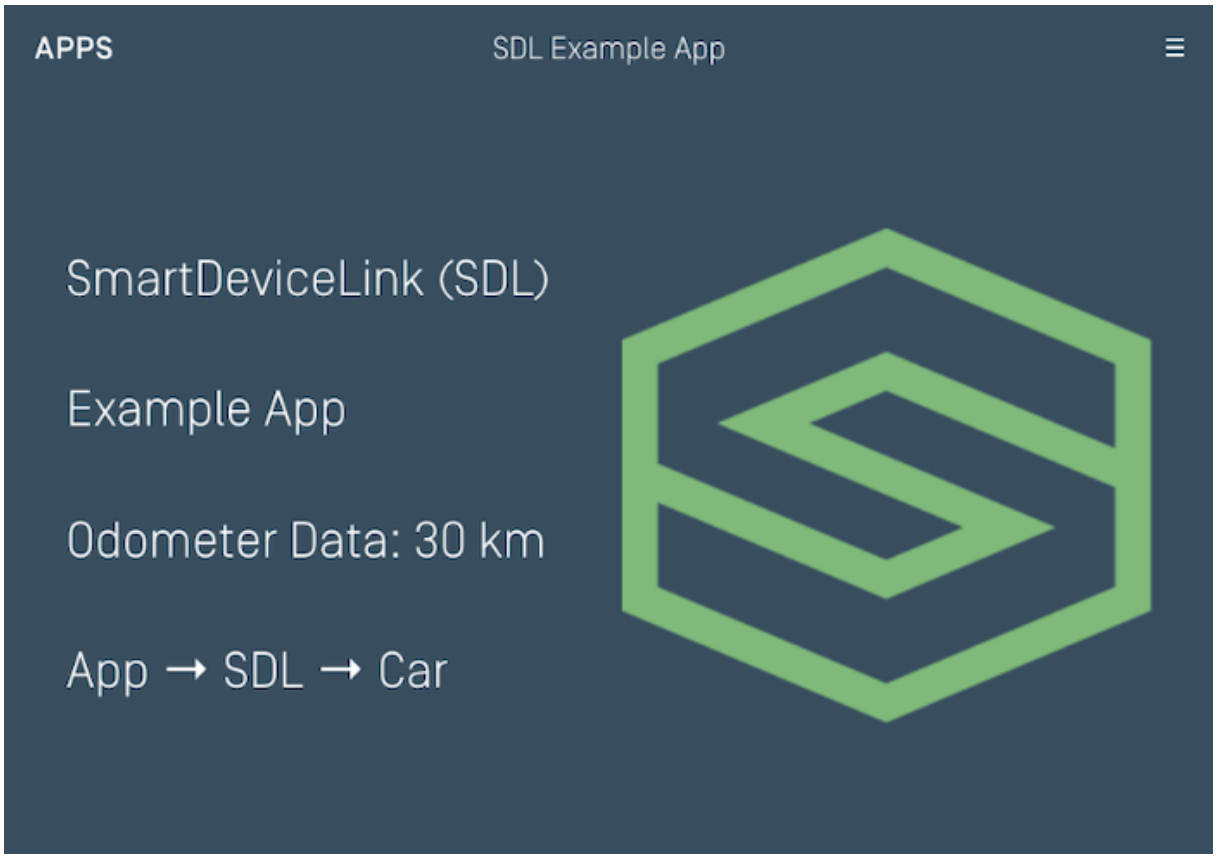
MEDIA (WITH A PROGRESS BAR)


APPS
Livio Music
JOHN PRINE
German Afternoons
John Prine
Linda Goes to Mars
German Afternoons
00:01:49 / 00:03:06

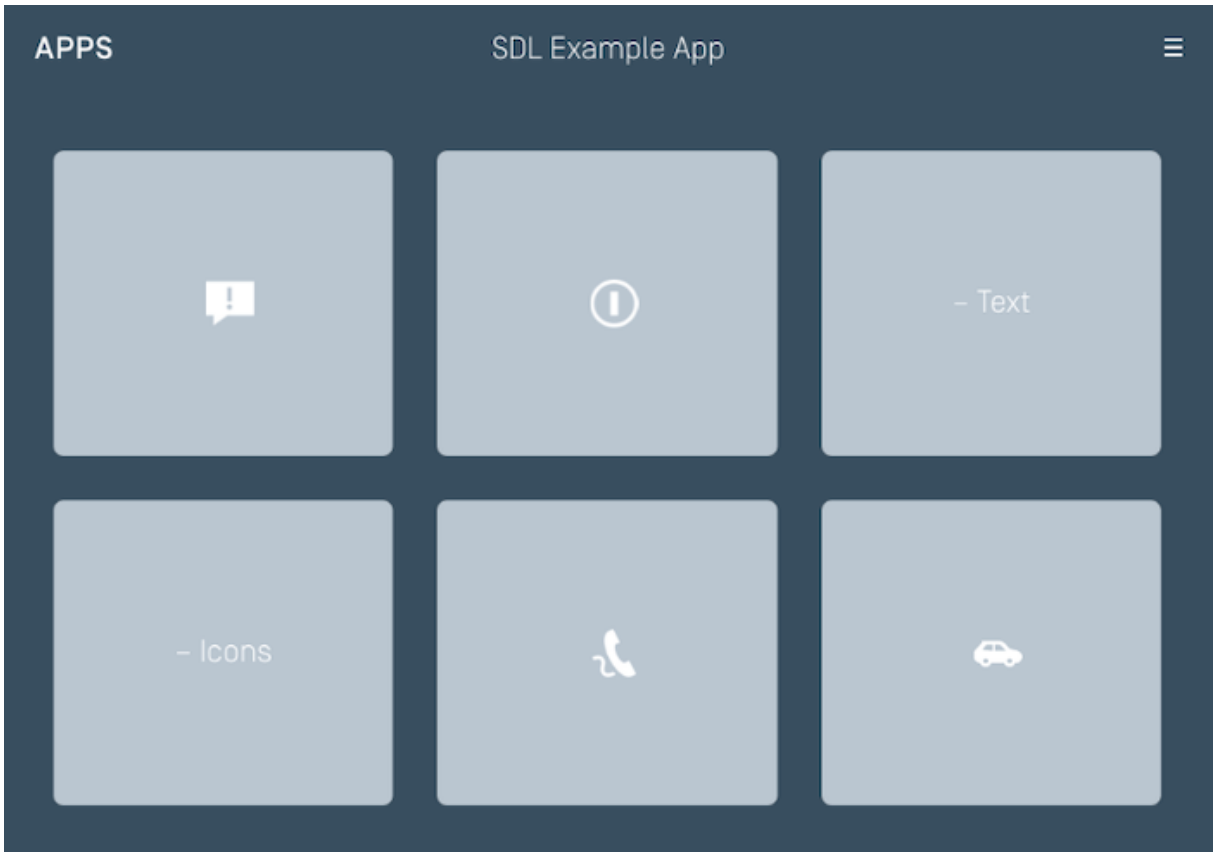
NON-MEDIA

GRAPHIC WITH TEXT

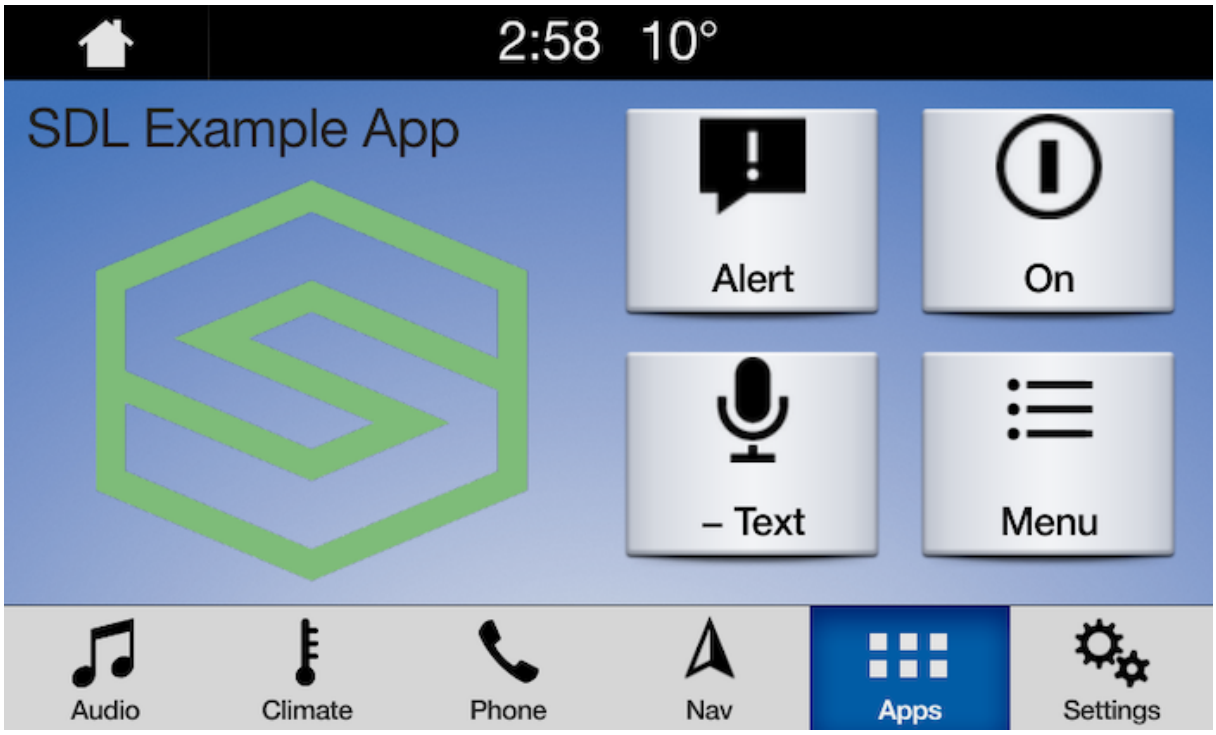
TEXT WITH GRAPHIC


APPS
SDL Example App
SmartDeviceLink (SDL)
Example App
Odometer Data: 30 km
App → SDL → Car

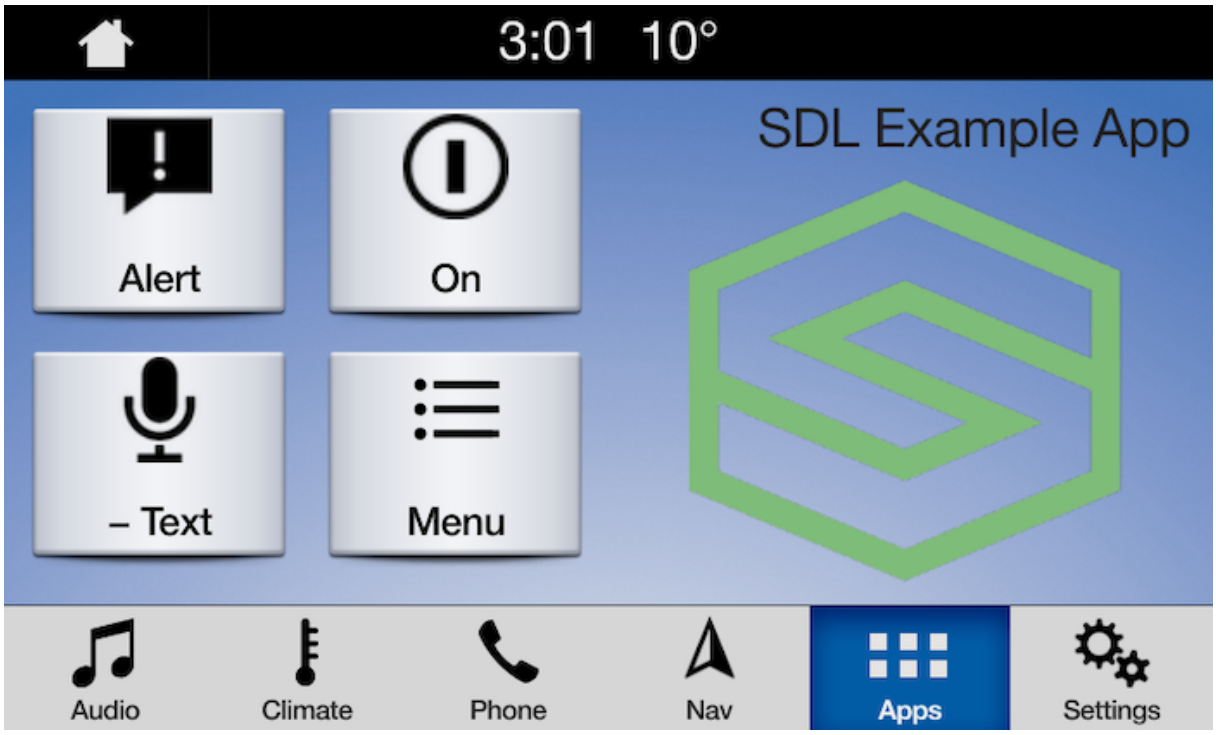
TILES ONLY



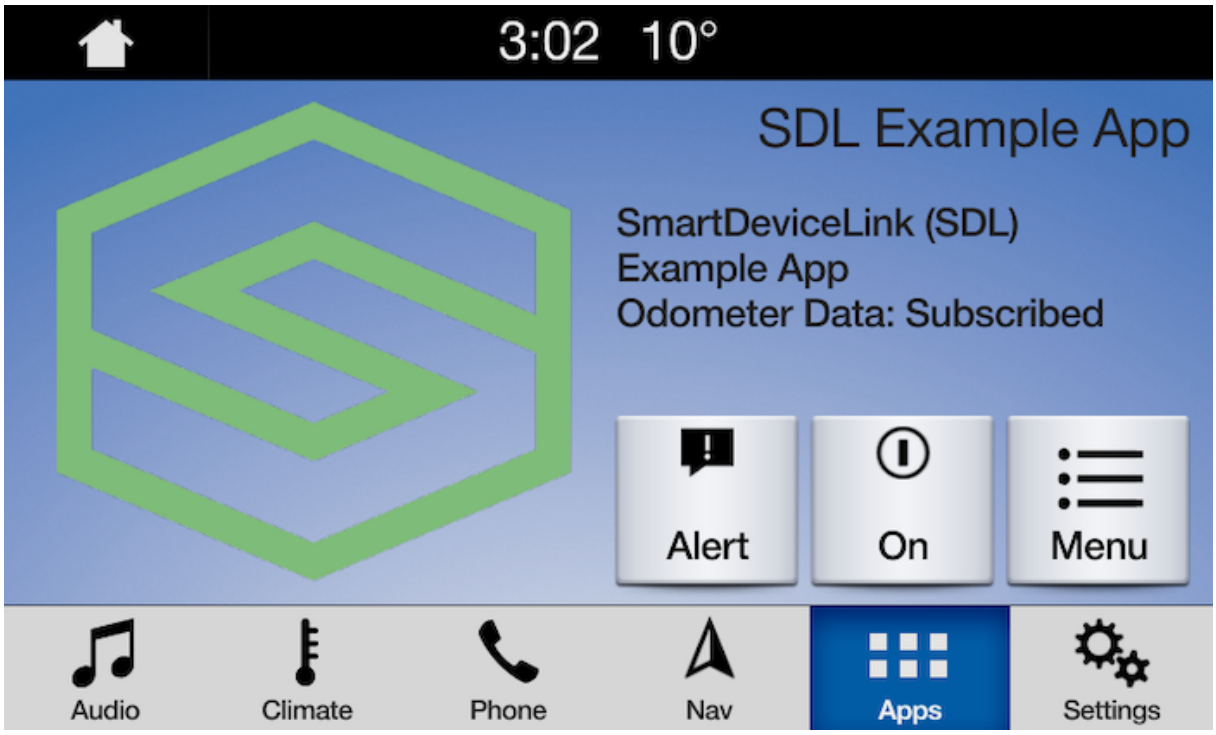
GRAPHIC WITH TILES



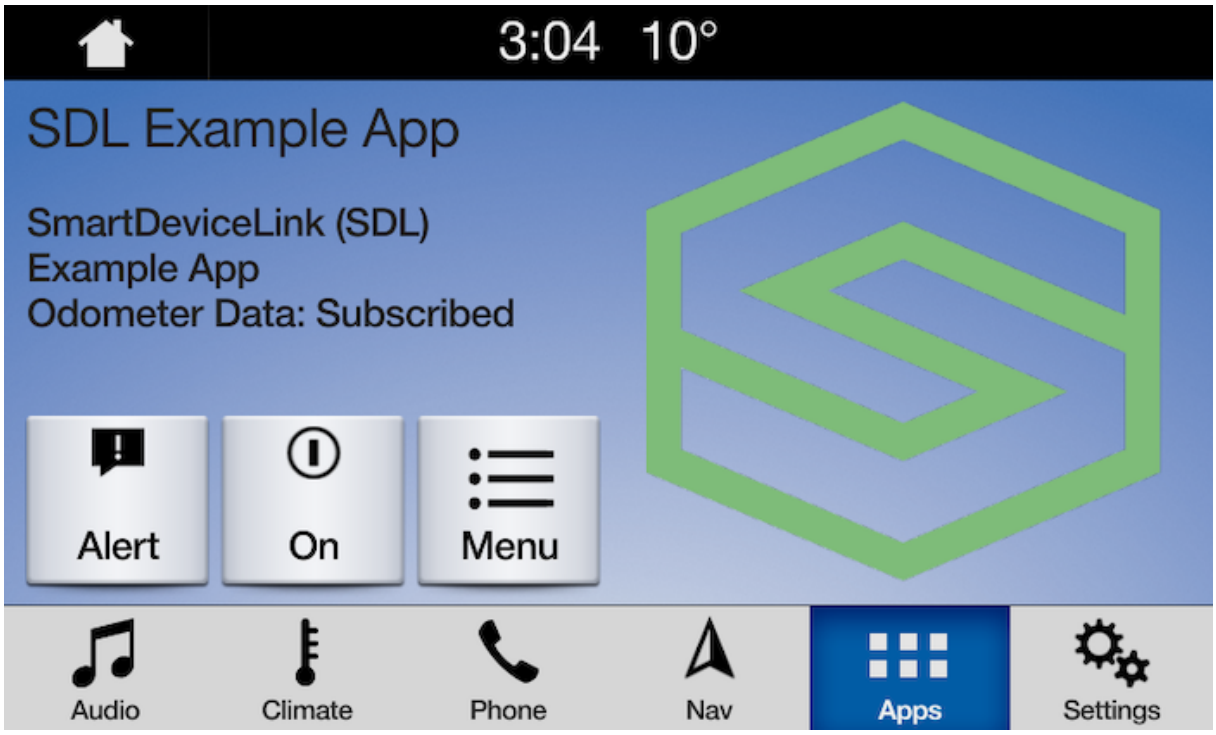
TILES WITH GRAPHIC



GRAPHIC WITH TEXT AND SOFT BUTTONS

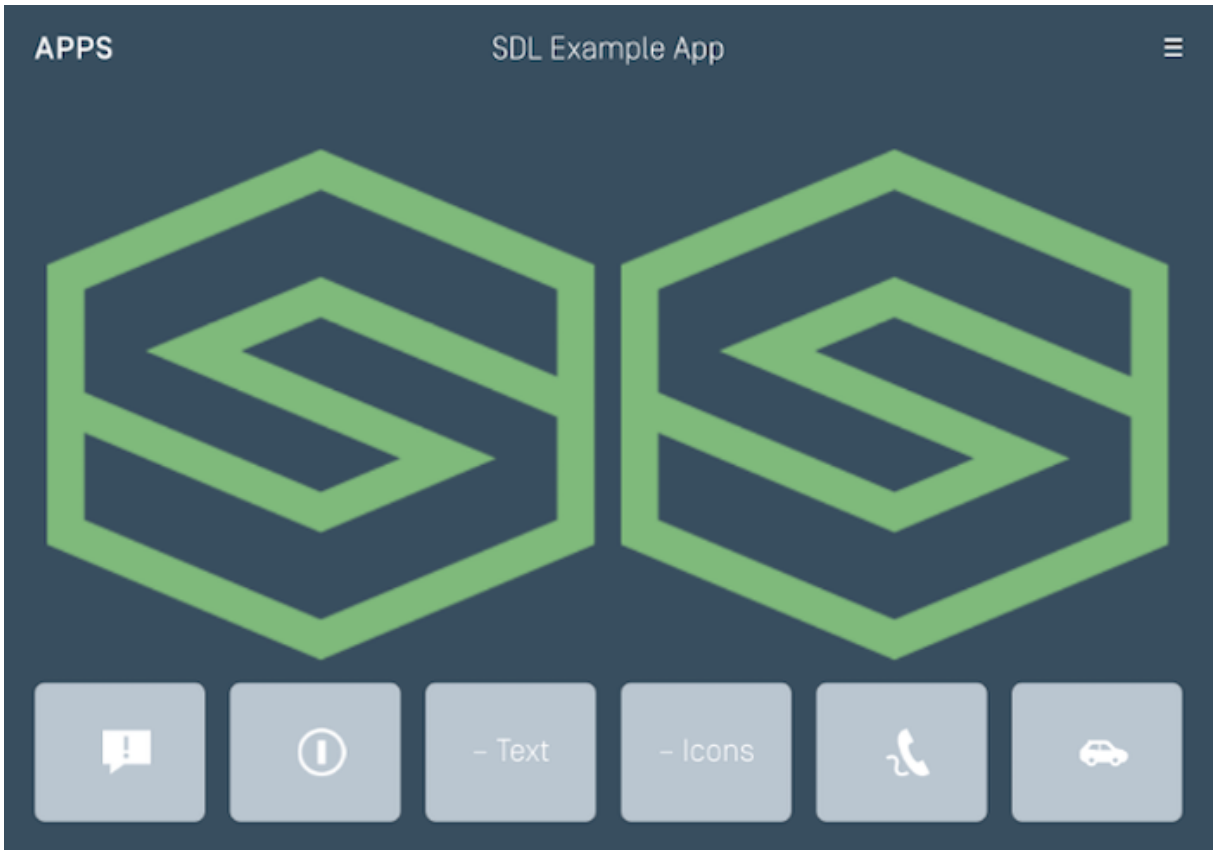


TEXT AND SOFT BUTTONS WITH GRAPHIC

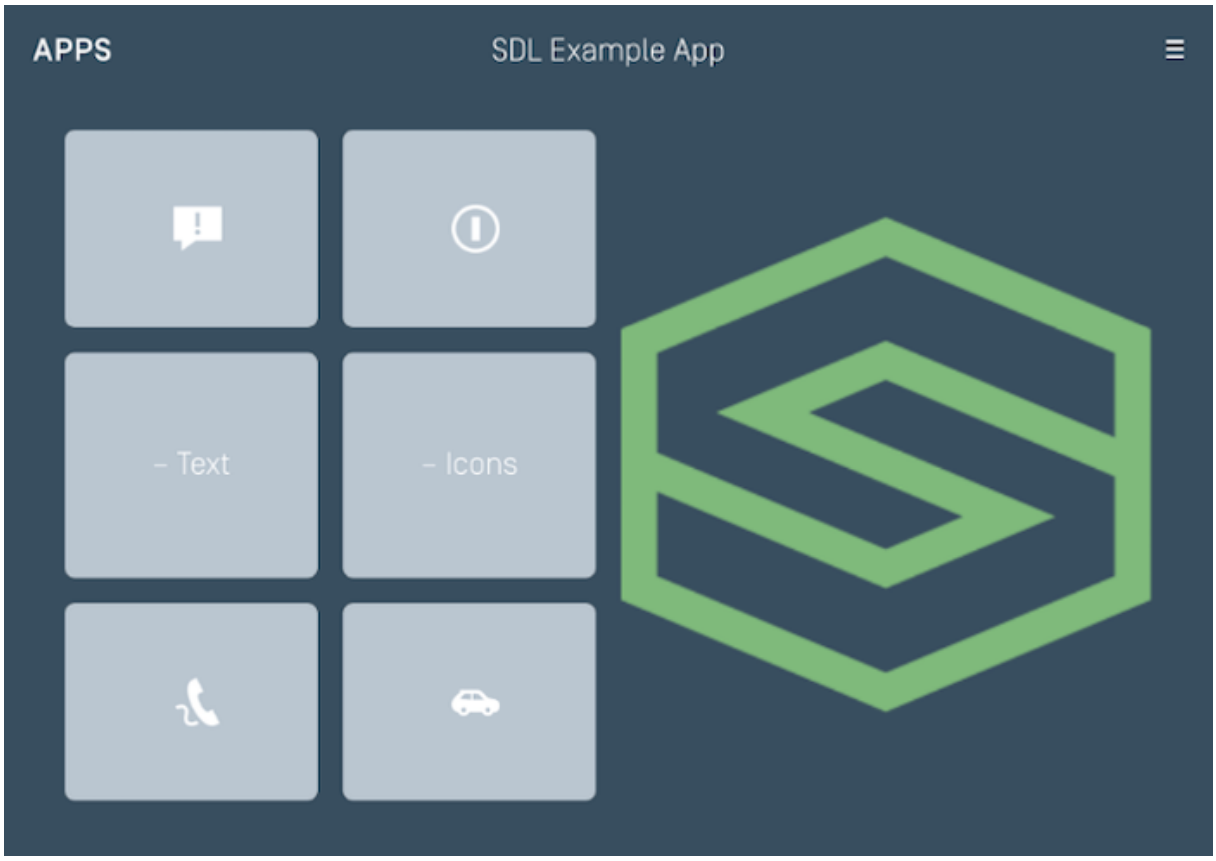


GRAPHIC WITH TEXT BUTTONS

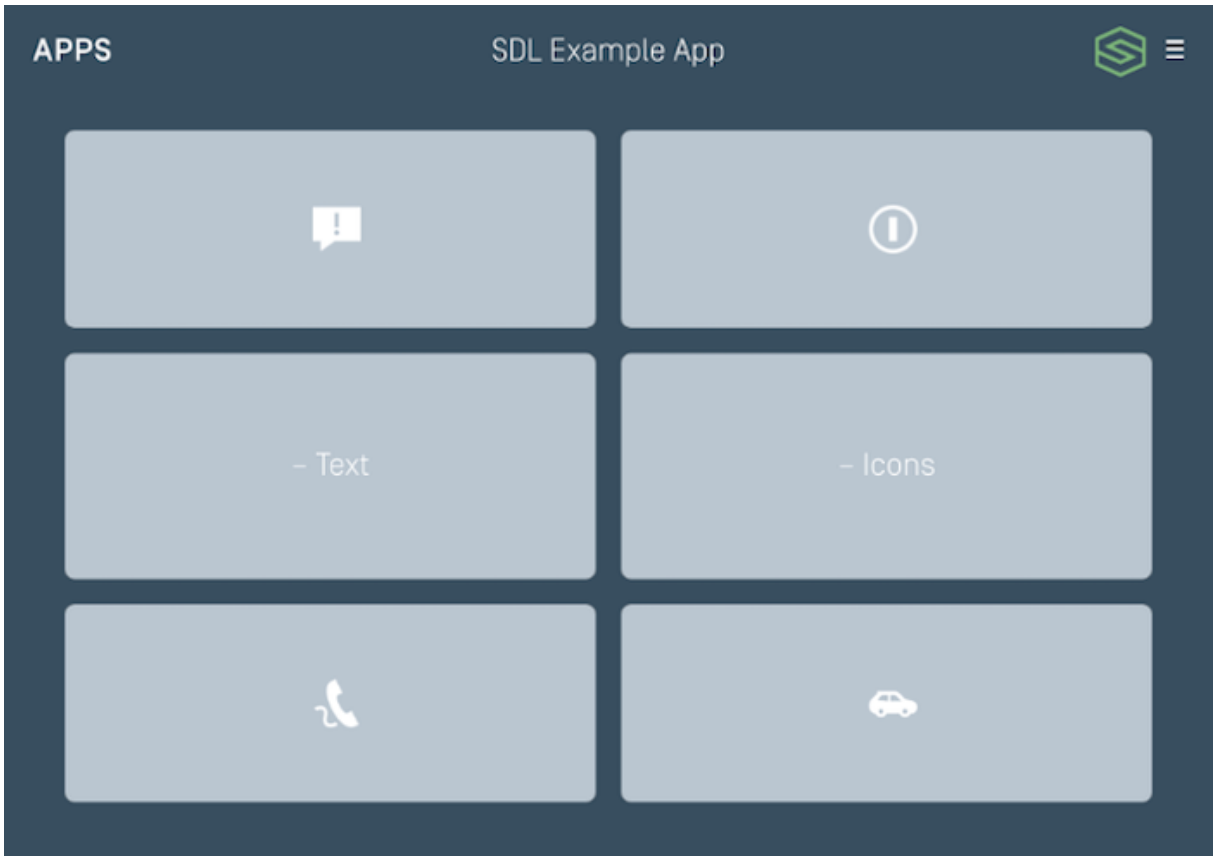
DOUBLE GRAPHIC WITH SOFT BUTTONS



TEXT BUTTONS WITH GRAPHIC



TEXT BUTTONS ONLY



LARGE GRAPHIC WITH SOFT BUTTONS

LARGE GRAPHIC ONLY

Template Text

You can easily display text, images, and buttons using the `SDLScreenManager`. To update the UI, simply give the manager your new data and (optionally) sandwich the update between the manager's `beginUpdates` and `endUpdatesWithCompletionHandler` methods.

Text Fields

SDLSCREENMANAGER PARAMETER NAME	DESCRIPTION
textField1	The text displayed in a single-line display, or in the upper display line of a multi-line display
textField2	The text displayed on the second display line of a multi-line display
textField3	The text displayed on the third display line of a multi-line display
textField4	The text displayed on the bottom display line of a multi-line display
mediaTrackTextField	The text displayed in the in the track field; this field is only valid for media applications
textAlignment	The text justification for the text fields; the text alignment can be left, center, or right
textField1Type	The type of data provided in textField1
textField2Type	The type of data provided in textField2
textField3Type	The type of data provided in textField3
textField4Type	The type of data provided in textField4
title	The title of the displayed template

Showing Text

OBJC | SWIFT

Removing Text

To remove text from the screen simply set the screen manager property to `nil`.

OBJC | SWIFT

Template Images

You can easily display text, images, and buttons using the `SDLScreenManager`. To update the UI, simply give the manager your new data and (optionally) sandwich the update between the manager's `beginUpdates` and `endUpdatesWithCompletionHandler` methods.

Image Fields

SDLSCREENMANAGER PARAMETER NAME	DESCRIPTION
primaryGraphic	The primary image in a template that supports images
secondaryGraphic	The second image in a template that supports multiple images

Showing Images

Creating an SDLArtwork

Create an `SDLArtwork` object which can be manually uploaded or set into the `SDLScreenManager` and automatically uploaded. An `SDLArtwork` includes information about

whether the image should be persisted between vehicle startups, whether the image is a template image and should be re-colored, and more.

OBJC | SWIFT

Setting Primary Graphic

OBJC | SWIFT

Removing Images

To remove an image from the screen you just need to set the screen manager property to `nil`.

OBJC | SWIFT

Overwriting Images

When a file is to be uploaded to the module, the library checks if a file with the same name has already been uploaded to module and skips the upload if it can. For cases where an image by the same name needs to be re-uploaded, the `SDLArtwork` / `SDLFile`'s `overwrite` property should be used. Setting `overwrite` to `true` before passing the image to a `SDLScreenManager` method such as `primaryGraphic` and `secondaryGraphic` will force the image to be re-uploaded. This includes methods such as `preloadChoices:withCompletionHandler:` where the arguments passed in contain images.

NOTE

Please note that many production modules on the road do not refresh the HMI with the new image if the file name has not changed. If you want the image to refresh on the screen immediately, we suggest using two image names and toggling back and forth between the names each time you update the image.

This issue may also extend to menus, alerts, and other UI features even if they're not on-screen at the time. Because of these issues, we do not recommend that you try to overwrite an image. Instead, you can delete an image file using the `SDLFileManager` and re-upload it once the deletion completes, or you may use a different file name.

Templating Images (RPC v5.0+)

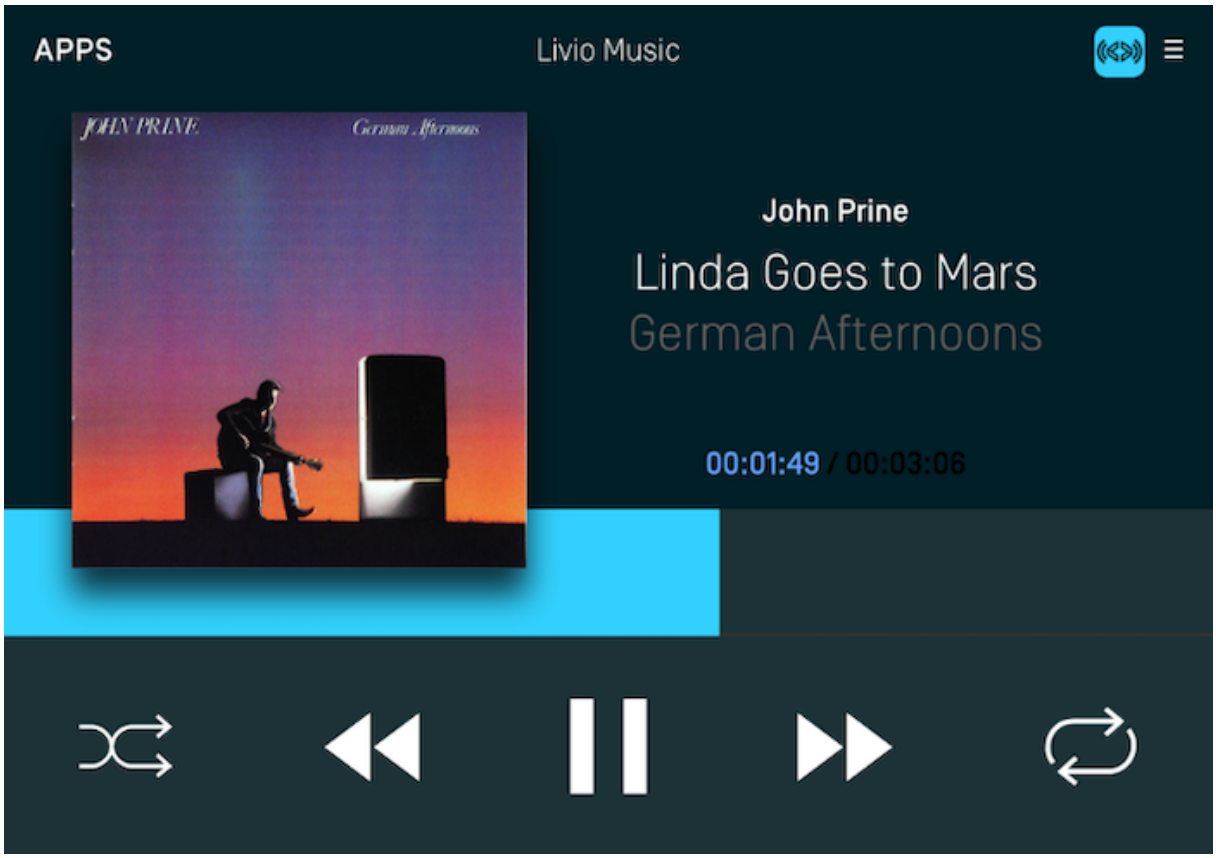
Templated images are tinted by Core so the image is visible regardless of whether your user has set the head unit to day or night mode. For example, if a head unit is in night mode with a dark theme (see Customizing the Template section for more details on how to customize theme colors), then your templated images will be displayed as white. In the day theme, the image will automatically change to black.

Soft buttons, menu icons, and primary / secondary graphics can all be templated. A template image works very much like it does on iOS and in fact, it uses the same API as iOS. Any `SDLArtwork` created with a `UIImage` that has a `renderingMode` of `alwaysTemplate` will be templated via SDL as well. Images that you wish to template must be PNGs with a transparent background and only one color for the icon. Therefore, templating is only useful for things like icons and not for images that must be rendered in a specific color.

Templated Images Example

In the screenshots below, the shuffle and repeat icons have been templated. In night mode, the icons are tinted white and in day mode the icons are tinted black.

NIGHT MODE



DAY MODE

OBJC | SWIFT

Static Icons

Static icons are pre-existing images on the remote system that you may reference and use in your own application. Each OEM will design their own custom static icons but you can get an overview of the available icons from the icons designed for the open source Generic HMI. Static icons are fully supported by the screen manager via an `SDLArtwork` initializer. Static icons can be used in primary and secondary graphic fields, soft button image fields, and menu icon fields.

OBJC | SWIFT

Template Custom Buttons

You can easily create and update custom buttons (called Soft Buttons in SDL) using the `SDLScreenManager`. To update the UI, simply give the manager your new data and (optionally) sandwich the update between the manager's `beginUpdates` and `endUpdatesWithCompletionHandler` methods.

Soft Button Fields

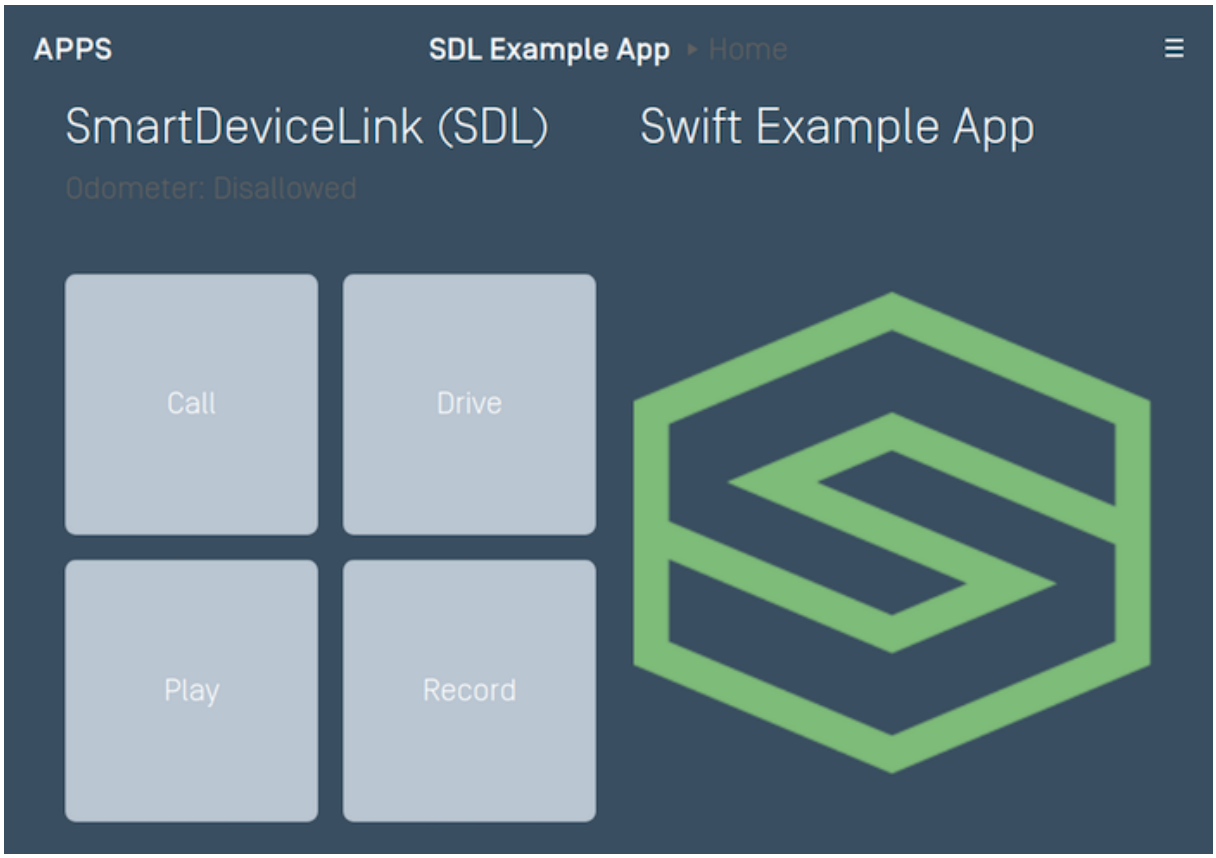
SDLSCREENMANAGER PARAMETER NAME	DESCRIPTION
softButtonObjects	An array of buttons. Each template supports a different number of soft buttons

Creating Soft Buttons

To create a soft button using the `SDLScreenManager`, you only need to create a custom name for the button and provide the text for the button's label and/or an image for the button's icon. If your button cycles between different states (e.g. a button used to set the repeat state of a song playlist can have three states: repeat-off, repeat-one, and repeat-all), you can create all the states on initialization.

There are three different ways to create a soft button: with only text, with only an image, or with both text and an image. If creating a button with an image, we recommend that you template the image so its color works well with both the day and night modes of the head unit. For more information on templating images please see the Template Images guide.

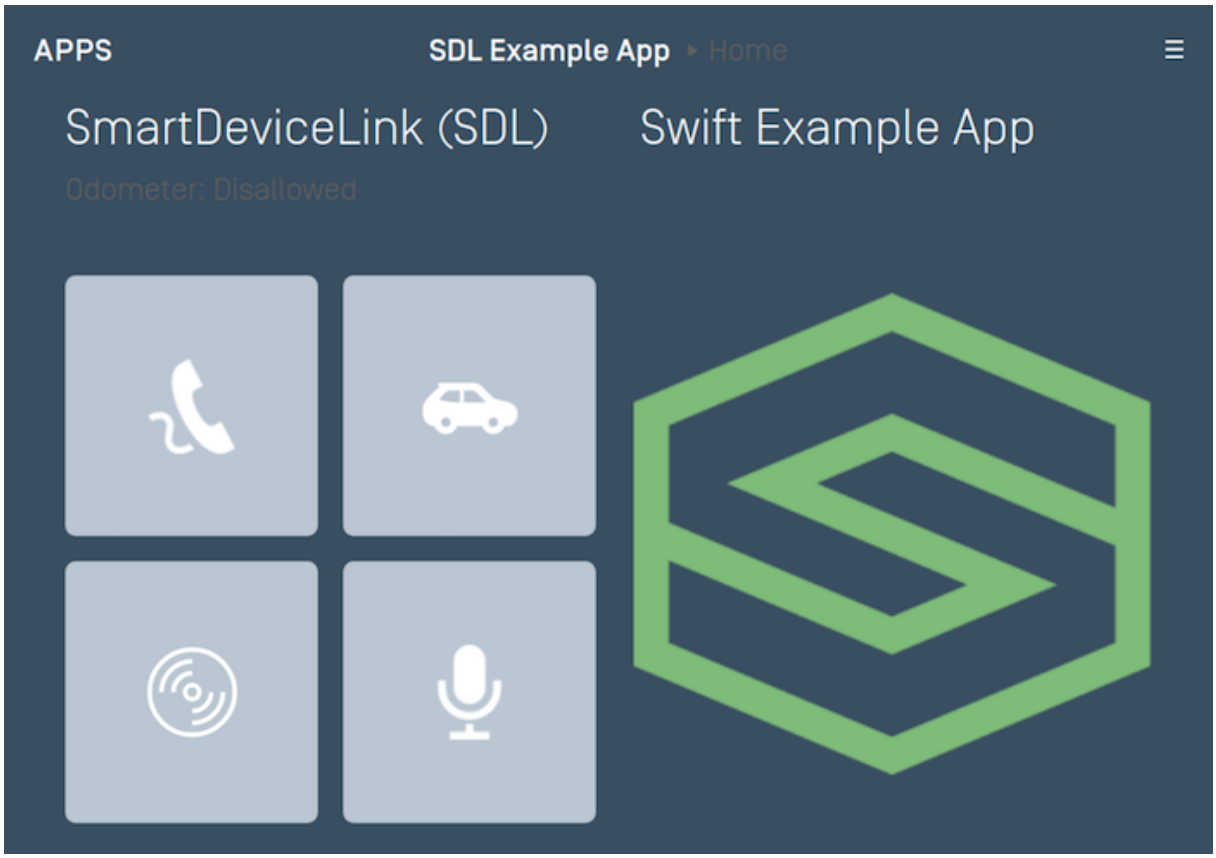
Text Only Soft Buttons



OBJC | SWIFT

Image Only Soft Buttons

You can use the `SDLSystemCapabilityManager` to check if the HMI supports soft buttons with images. If you send image-only buttons to a HMI that does not support images, then the library will not send the buttons as they will be rejected by the head unit. If all your soft buttons have text in addition to images, the library will send the text-only buttons if the head unit does not support images.

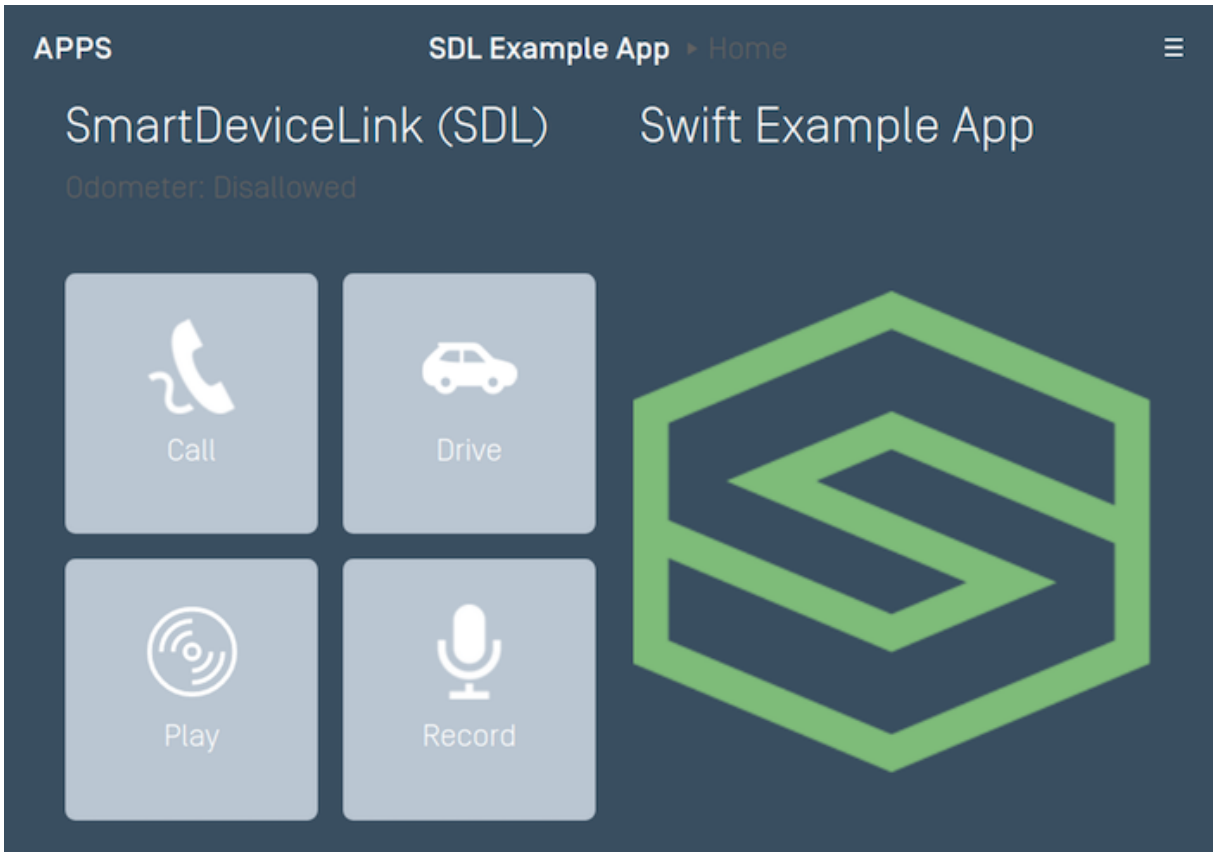


OBJC | SWIFT

Once you know that the HMI supports images in soft buttons you can create and send the image-only soft buttons.

OBJC | SWIFT

Image and Text Soft Buttons



OBJC | SWIFT

Highlighting a Soft Button

When a button is highlighted its background color will change to indicate that it has been selected.

HIGHLIGHT ON

HIGHLIGHT OFF

OBJC | SWIFT

Updating Soft Button States

When the soft button state needs to be updated, simply tell the `SoftButtonObject` to transition to the next state. If your button states do not cycle in a predictable order, you can also tell the soft button which state to transition to by passing the `stateName` of the new soft button state.

OBJC | SWIFT

Deleting Soft Buttons

To delete soft buttons, simply pass the screen manager a new array of soft buttons. To delete all soft buttons, simply pass the screen manager an empty array.

OBJC | SWIFT

Using RPCs

You can also send soft buttons manually using the `Show` RPC. Note that if you do so, you must not mix the `SDLScreenManager` soft buttons and manually sending the `Show` RPC. Additionally, the `SDLScreenManager` takes soft button ids 0 - 10000. Ensure that if you use custom RPCs, that the soft button ids you use are outside of this range.

Template Subscription Buttons

This guide shows you how to subscribe and react to "subscription" buttons. Subscription buttons are used to detect when the user has interacted with buttons located in the car's center console or steering wheel. A subscription button may also show up as part of your template, however, the text and/or image used in the button is determined by the template and is (usually) not customizable.

In the screenshot below, the pause, seek left and seek right icons are subscription buttons. Once subscribed to, for example, the seek left button, you will be notified when the user selects the seek left button on the HMI or when they select the seek left button on the car's center console and/or steering wheel.

Types of Subscription Buttons

There are three general types of subscriptions buttons: audio related buttons only used for media apps, navigation related buttons only used for navigation apps, and general buttons, like preset buttons and the OK button, that can be used with all apps. Please note that if your app type is not `MEDIA` or `NAVIGATION`, your attempt to subscribe to media-only or navigation-only buttons will be rejected.

BUTTON	APP TYPE	RPC VERSION
Ok	All	v1.0+
Preset 0-9	All	v1.0+
Search	All	v1.0+
Play / Pause	Media only	v5.0+
Seek left	Media only	v1.0+
Seek right	Media only	v1.0+
Tune up	Media only	v1.0+
Tune down	Media only	v1.0+
Center Location	Navigation only	v6.0+
Zoom In	Navigation only	v6.0+
Zoom Out	Navigation only	v6.0+
Pan Up	Navigation only	v6.0+
Pan Up-Right	Navigation only	v6.0+
Pan Right	Navigation only	v6.0+
Pan Down-Right	Navigation only	v6.0+
Pan Down	Navigation only	v6.0+

BUTTON	APP TYPE	RPC VERSION
Pan Down-Left	Navigation only	v6.0+
Pan Left	Navigation only	v6.0+
Pan Up-Left	Navigation only	v6.0+
Toggle Tilt	Navigation only	v6.0+
Rotate Clockwise	Navigation only	v6.0+
Rotate Counter-Clockwise	Navigation only	v6.0+
Toggle Heading	Navigation only	v6.0+

Subscribing to Subscription Buttons

You can easily subscribe to subscription buttons using the `SDLScreenManager`. Simply tell the manager which button to subscribe and you will be notified when the user selects the button.

There are two different ways to receive button press notifications. The first is to pass a block handler that will get called when the button is selected. The second is to pass a selector that will be notified when the button is selected.

Subscribe with a Block Handler

Once you have subscribed to the button with a block handler, the handler will be called whenever the button has been selected. If an error occurs attempting to subscribe to the button, the error will be returned in the `error` parameter.

OBJC | SWIFT

Subscribe with a Selector

Once you have subscribed to the button, the selector will be called when the button has been selected. If there is an error subscribing to the subscribe button it will be returned in the `error` parameter.

The selector can be created with between zero and four parameters of types in the following order: `SDLButtonName`, `NSError`, `SDLOnButtonPress`, and `SDLOnButtonEvent`. When the fourth parameter, `SDLOnButtonEvent`, is omitted from the selector, then you will only be notified when a button press occurs. When the third parameter, `SDLOnButtonPress` is omitted from the selector, you will be unable to distinguish between short and long button presses.

OBJC | SWIFT

OBJC | SWIFT

Unsubscribing from Subscription Buttons

When unsubscribing, you will need to pass the observer object and which button name that you want to unsubscribe. If you subscribed using a handler, use the observer object returned when you subscribed. If you subscribed using a selector, use the same observer object you passed when subscribing.

OBJC | SWIFT

Media Buttons

The play/pause, seek left, seek right, tune up, and tune down subscribe buttons can only be used if the app type is `MEDIA`. Depending on the OEM, the subscribed button could show up as an on-screen button in the `MEDIA` template, work as a physical button on the car console or steering wheel, or both. For example, Ford's SYNC® 3 HMI will add the

play/pause, seek right, and seek left soft buttons to the media template when you subscribe to those buttons. However, those buttons will also trigger when the user uses the seek left / seek right buttons on the steering wheel.

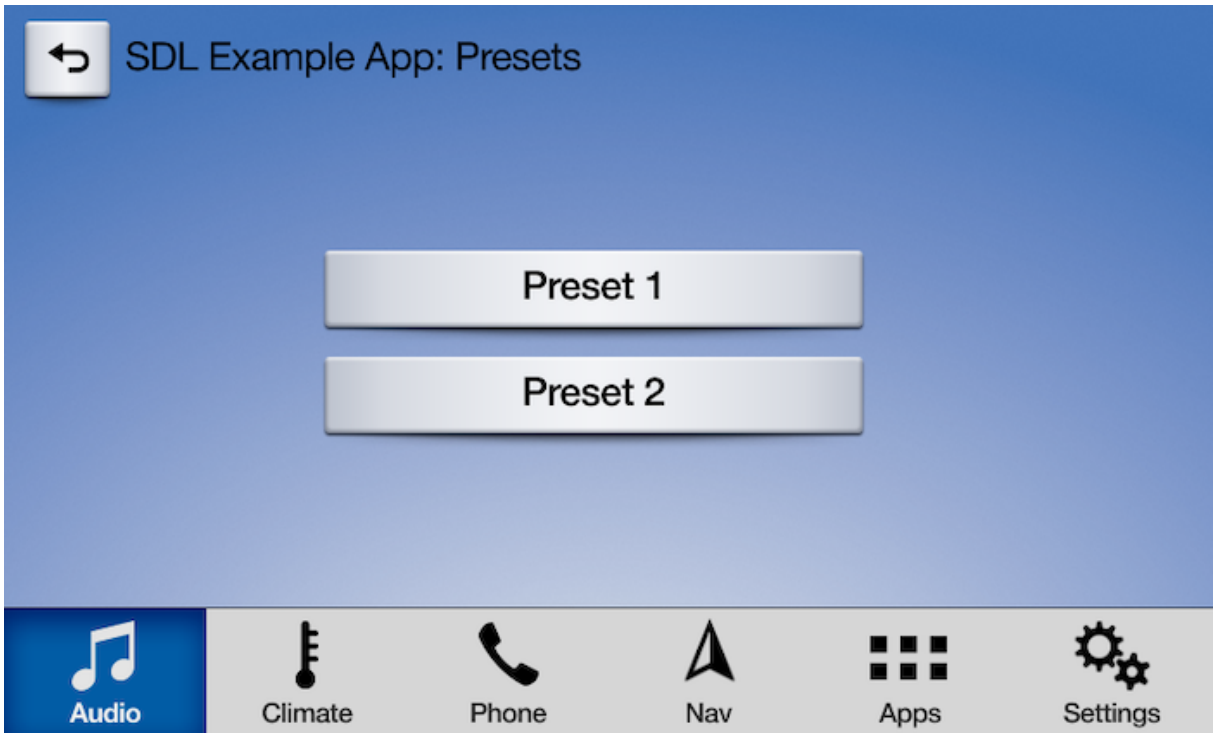
If desired, you can change the style of the play/pause button image between a play, stop, or pause icon by updating the audio streaming indicator, and you can also set the style of the next/previous buttons between a track or time seek style. See the Media Clock guide for more information.

> **NOTE**
>
> Before library v.6.1 and RPC v5.0, `Ok` and `PlayPause` were combined into `Ok`. Subscribing to `Ok` will, in v6.1+, also subscribe you to `PlayPause`. This means that for the time being, *you should not simultaneously subscribe to `Ok` and `PlayPause`*. In a future major version, this will change. For now, only subscribe to either `Ok` or `PlayPause` and the library will execute the right action based on the connected head unit.

OBJC | SWIFT

Preset Buttons

All app types can subscribe to preset buttons. Depending on the OEM, the preset buttons may be added to the template when subscription occurs. Preset buttons can also be physical buttons on the console that will notify the subscriber when selected. An OEM may support only template buttons or only hard buttons or they may support both template and hard buttons. The screenshot below shows how the Ford SYNC® 3 HMI displays the preset buttons on the HMI.



Checking if Preset Buttons are Supported

You can check if a HMI supports subscribing to preset buttons, and if so, how many preset buttons are supported, by checking the system capability manager.

OBJC | SWIFT

Subscribing to Preset Buttons

OBJC | SWIFT

OBJC | SWIFT

Navigation Buttons

Head units supporting RPC v6.0+ may support subscription buttons that allow your user to drag and scale the map using hard buttons located on car's center console or steering wheel. Subscriptions to navigation buttons will only succeed if your app's type is `NAVIGATION`. If subscribing to these buttons succeeds, you can remove any buttons of your own from your map screen. If subscribing to these buttons fails, you can display buttons of your own on your map screen.

Subscribing to Navigation Buttons

OBJC | SWIFT

Main Menu

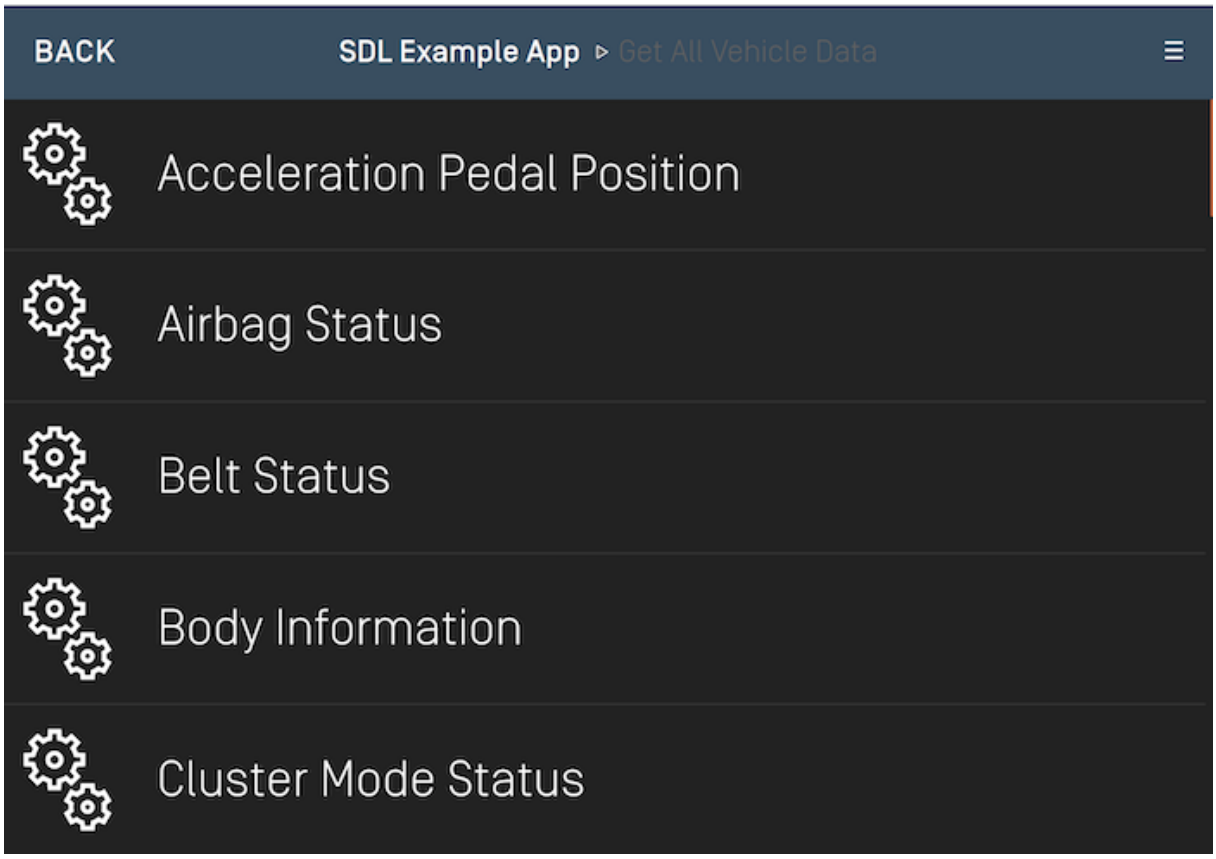
You have two different options when creating menus. One is to simply add items to the default menu available in every template. The other is to create a custom menu that pops up when needed. You can find more information about these popups in the Popup Menus section. This guide will cover using the default menu / menu button.

> **NOTE**
>
> Every template has a main menu button. The position of this button varies between templates and cannot be removed from the template. Some OEMs may format certain templates to not display the main menu button if you have no menu items (such as the navigation map view).

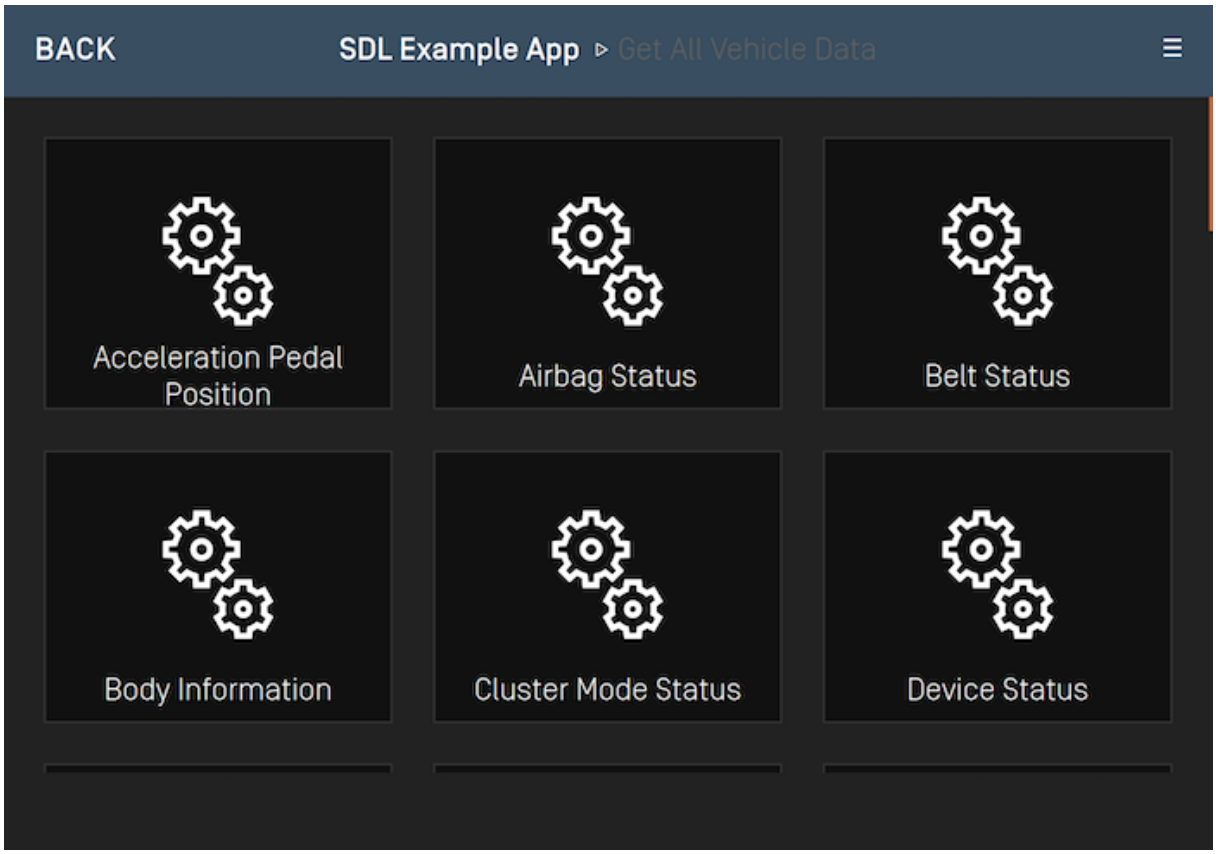
Setting the Menu Layout (RPC v6.0+)

On some newer head units, you may have the option to display menu items as a grid of tiles instead of the default list layout. To determine if the head unit supports the tiles layout, check the `SystemCapabilityManager`'s `defaultMainWindowCapability.menuLayoutsAvailable` property after successfully connecting to the head unit. To set the menu layout using the screen manager, you will need to set the `ScreenManager.menuConfiguration` property.

LIST MENU LAYOUT



GRID MENU LAYOUT

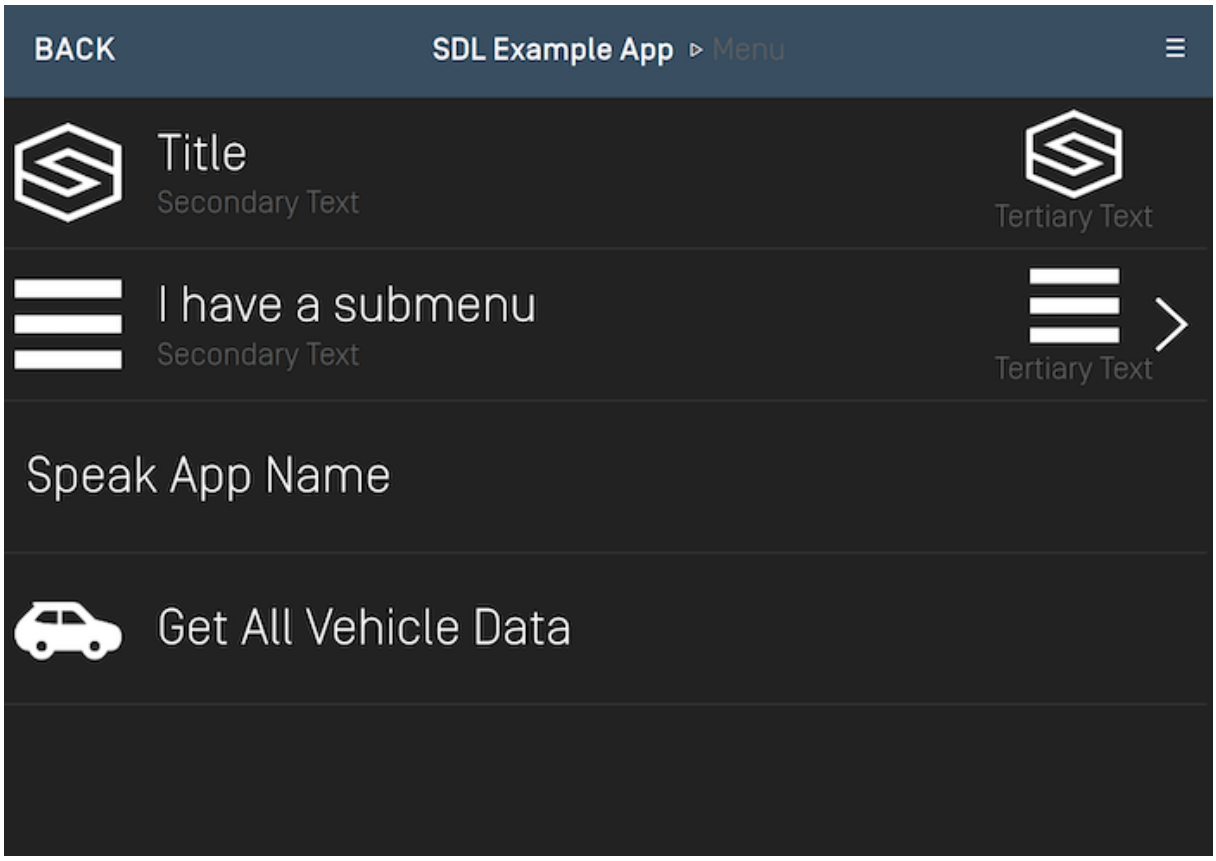


OBJC | SWIFT

Adding Menu Items

The best way to create and update your menu is to the use the Screen Manager API. The screen manager contains two menu related properties: `menu`, and `voiceCommands`. Setting an array of `SDLMenuCell`s into the `menu` property will automatically set and update your menu and submenus, while setting an array of `SDLVoiceCommand`s into the `voiceCommands` property allows you to use "hidden" menu items that only contain voice recognition data. The user can then use the IVI system's voice engine to activate this command even though it will not be displayed within the main menu.

To find out more information on how to create `voiceCommands` see the related documentation.



NOTE

Head units supporting RPC v7.1+ may support displaying `secondaryText`, `tertiaryText`, and `secondaryArtwork`. This gives the user a richer experience by displaying more data. Attempting to set this data on head units that do not support RPC 7.1+ will result in that data not being displayed to the user.

To determine if the head unit supports displaying these fields, you can check the `SystemCapabilityManager`'s `defaultMainWindowCapability.textFields` / `defaultMainWindowCapability.imageFields` properties after successfully connecting to the head unit. Then check those arrays for objects with the related text / image field names.

OBJC | SWIFT

Adding Submenus

Adding a submenu is as simple as adding subcells to a `SDLMenuCell`. The submenu is automatically displayed when selected by the user. Currently menus only support one layer of subcells. In RPC v6.0+ it is possible to set individual submenus to use different layouts such as tiles or lists.

OBJC | SWIFT

Menu Item Artwork

Artworks will be automatically handled when using the screen manager API. First, a "non-artwork" menu will be displayed, then, when the artworks have finished uploading, the "artwork-ified" menu will be displayed. If you are doing this manually with RPCs, you will have to upload artworks using the file manager yourself and send the correct menu when they are ready.

Deleting and Changing Menu Items

The screen manager will intelligently handle deletions for you. If you want to show new menu items, simply set a new array of menu cells. If you want to have a blank menu, set an empty array. On supported systems, the library will calculate the optimal adds / deletes to create the new menu. If the system doesn't support this sort of dynamic updating, the entire list will be removed and re-added.

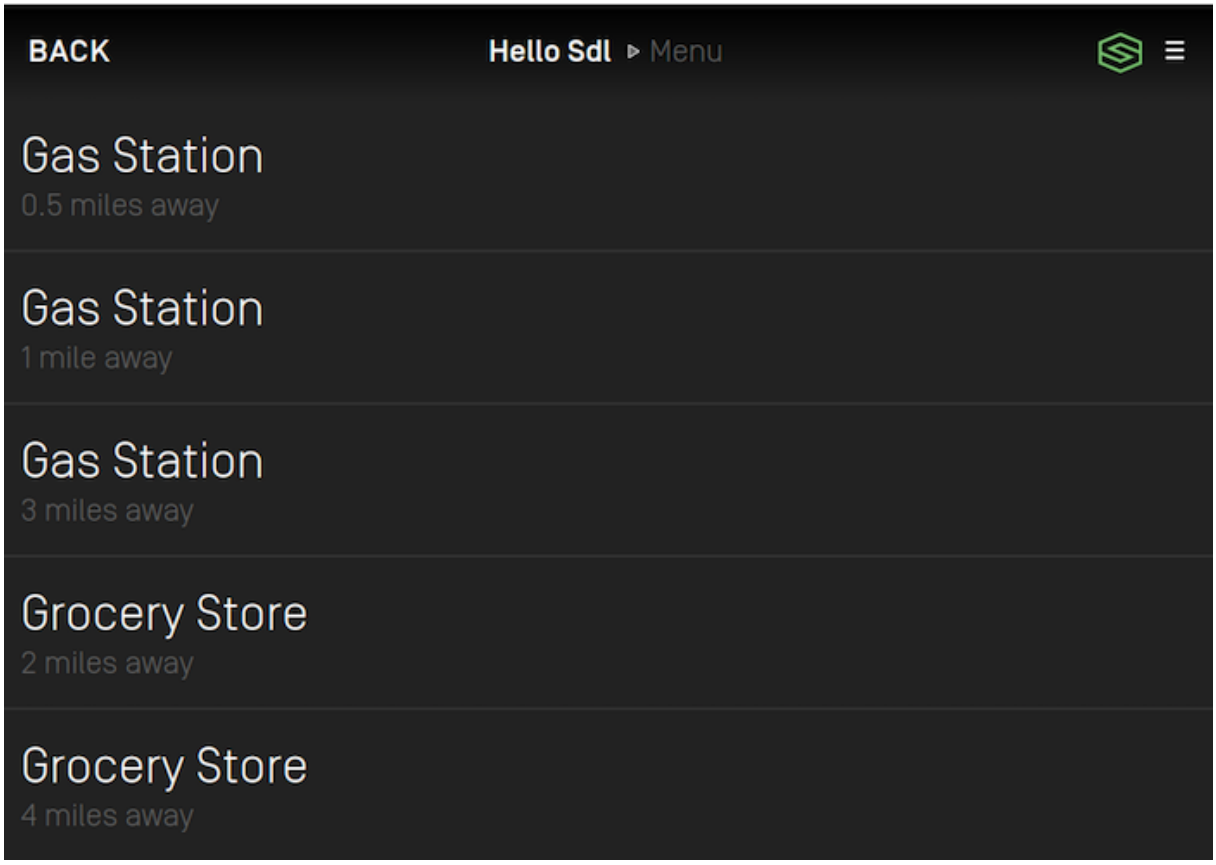
If you are doing this manually, you must use the `DeleteCommand` and `DeleteSubMenu` RPCs, passing the `cmdID`s you wish to delete.

Duplicate Menu Titles

Starting with SDL v7.1+ menu cells and sub-menu cells no longer require unique titles in order to be presented. For example, if you are trying to display points of interest as a list you can now have multiple locations with the same name but are not the same location. You cannot present multiple cells that are exactly the same. They must have some property that makes them different, such as `secondaryText` or an artwork.

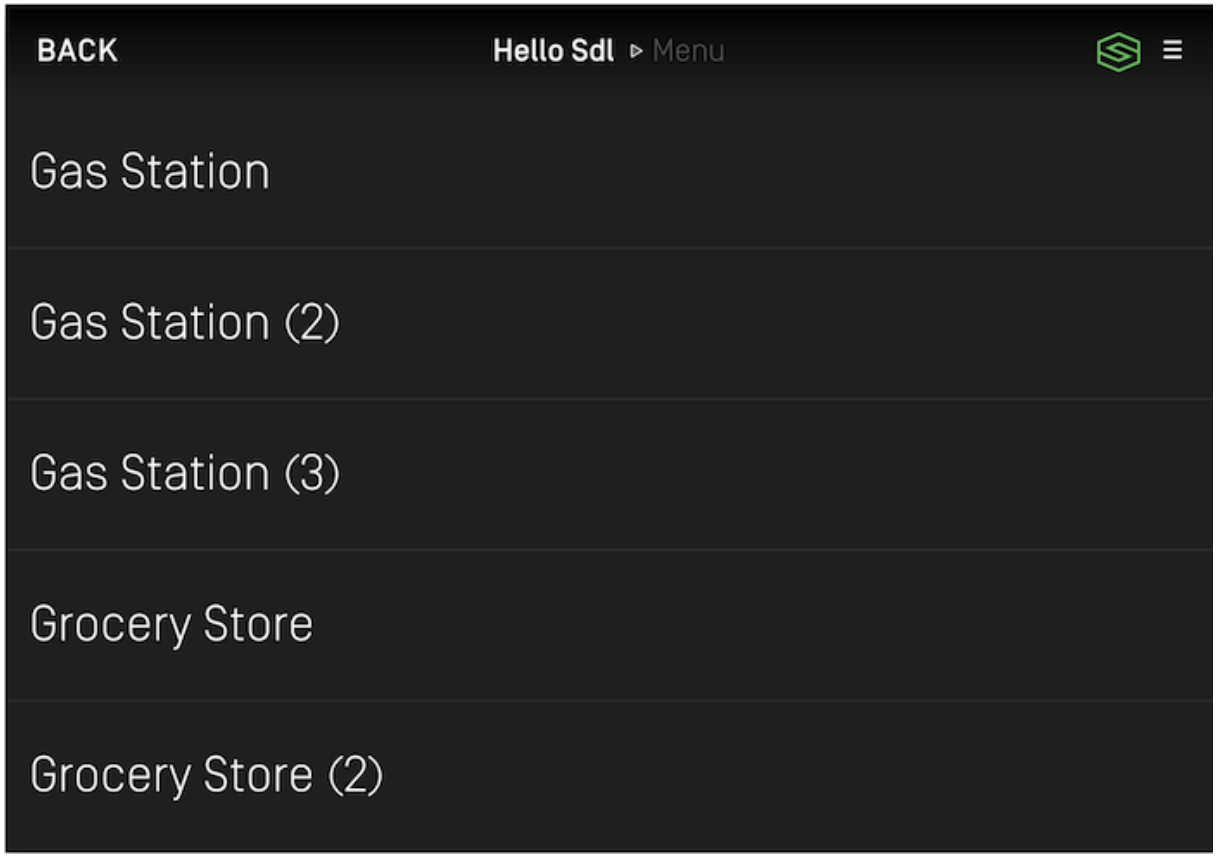
RPC V7.1+ CONNECTIONS

The titles on the menu will be displayed as provided even if there are duplicate titles.



RPC V7.0 AND BELOW CONNECTIONS

The titles on the menu will have a number appended to them when there are duplicate titles.



Using RPCs

The `AddCommand` RPC can be used to add items to the root menu or to a submenu. Each `AddCommand` RPC must be sent with a unique id, a voice-recognition command, and a set of menu parameters. The menu parameters include the menu name, the position of the item in the menu, and the id of the menu item's parent. If the menu item is being added to the root menu, then the parent id is 0. If it is being added to a submenu, then the parent id is the submenu's id.

To create a submenu using RPCs, you must use a `AddSubMenu` RPC with a unique id. When a response is received from the SDL Core, check if the submenu was added successfully. If it was, send an `AddCommand` RPC for each item in the submenu.

> **NOTE**
>
> You should not mix usage of the `SDLScreenManager` menu features and menu RPCs described above. You must use either one system or the other, but not both.

Popup Menus

SDL supports modal menus. The user can respond to the list of menu options via touch, voice (if voice recognition is supported by the head unit), or by keyboard input to search or filter the menu.

There are several UX considerations to take into account when designing your menus. The main menu should not be updated often and should act as navigation for your app. Popup menus should be used to present a selection of options to your user.

Presenting a Popup Menu

Presenting a popup menu is similar to presenting a modal view to request input from your user. It is possible to chain together menus to drill down, however, it is recommended to do so judiciously. Requesting too much input from a driver while they are driving is distracting and may result in your app being rejected by OEMs.

LAYOUT MODE	FORMATTING DESCRIPTION
Present as Icon	A grid of buttons with images
Present Searchable as Icon	A grid of buttons with images along with a search field in the HMI
Present as List	A vertical list of text
Present Searchable as List	A vertical list of text with a search field in the HMI

Creating Cells

An `SDLChoiceCell` is similar to a `UITableViewCell` without the ability to configure your own UI. We provide several properties on the `SDLChoiceCell` to set your data, but the layout itself is determined by the manufacturer of the head unit.

> **NOTE**
>
> On many systems, including VR commands will be *exponentially* slower than not including them. However, including them is necessary for a user to be able to respond to your prompt with their voice.

OBJC | SWIFT

Preloading Cells

If you know the content you will show in the popup menu long before the menu is shown to the user, you can "preload" those cells in order to speed up the popup menu

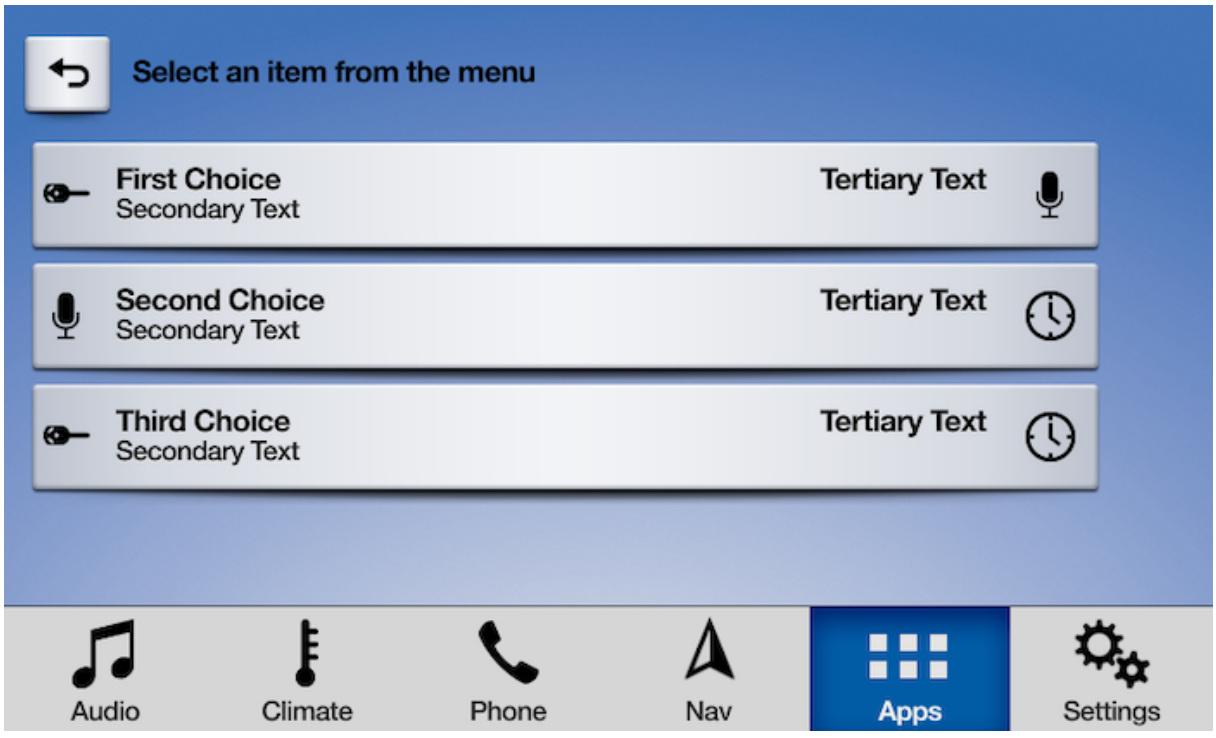
presentation at a later time. Once you preload a cell, you can reuse it in multiple popup menus without having to send the cell content to Core again.

OBJC | SWIFT

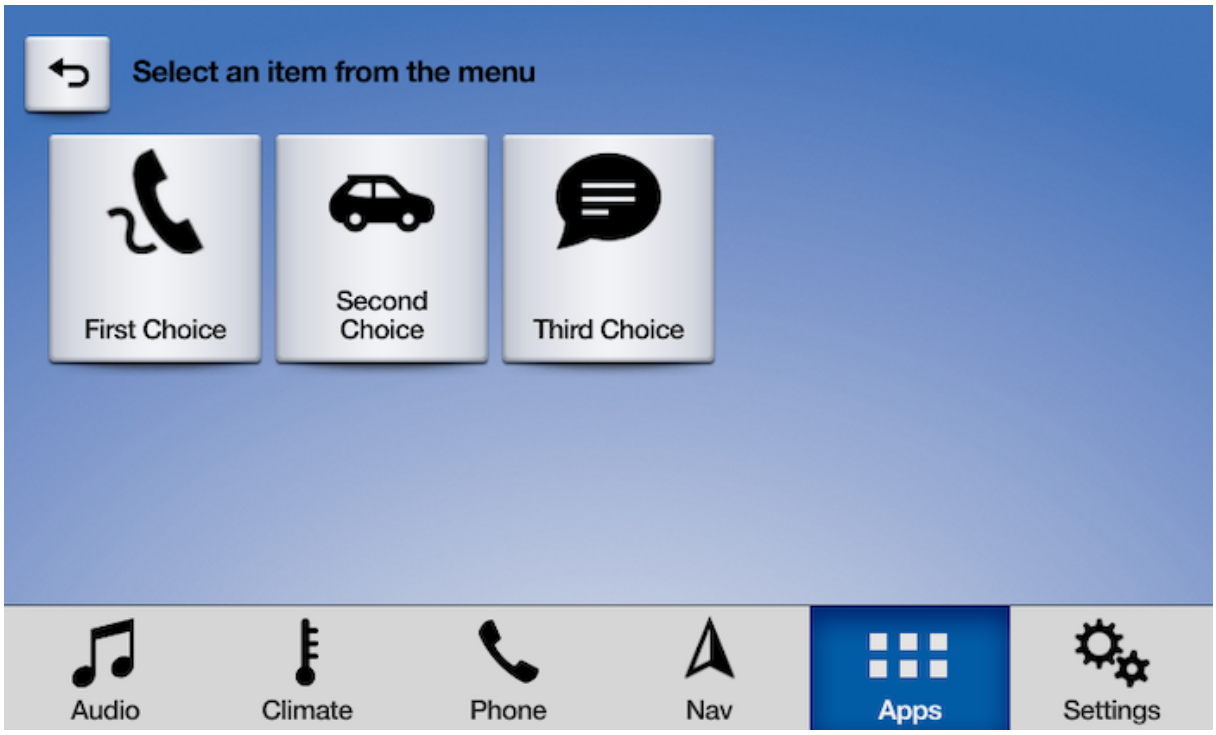
Presenting a Menu

To show a popup menu to the user, you must present the menu. If some or all of the cells in the menu have not yet been preloaded, calling the `present` API will preload the cells and then present the menu once all the cells have been uploaded. Calling `present` without preloading the cells can take longer than if the cells were preloaded earlier in the app's lifecycle especially if your cell has voice commands. Subsequent menu presentations using the same cells will be faster because the library will reuse those cells (unless you have deleted them).

MENU - LIST



MENU - ICON



> **NOTE**
>
> When you preload a cell, you **do not** need to maintain a reference to it. If you reuse a cell with the same properties that has already been preloaded (or previously presented), the cell will automatically be reused.

CREATING A CHOICE SET

In order to present a menu, you must bundle together a bunch of `SDLChoiceCell`s into an `SDLChoiceSet`.

> **NOTE**
>
> If the `SDLChoiceSet` contains an invalid set of `SDLChoiceCell`s, the initializer will return `nil`. This can happen, for example, if you have duplicate title text or if some, but not all choices have voice commands.

Some notes on various parameters (full documentation is available as API documentation on this website):

- Title: This is the title of the menu when presented
- Delegate: You must implement this delegate to receive callbacks based on the user's interaction with the menu
- Layout: You may present your menu as a set of tiles (like a `UICollectionView`) or a list (like a `UITableView`). If you are using tiles, it's recommended to use artworks on each item.

OBJC | SWIFT

IMPLEMENTING THE CHOICE SET DELEGATE

In order to present a menu, you must implement `SDLChoiceSetDelegate` in order to receive the user's input. When a choice is selected, you will be passed the `cell` that was selected, the manner in which it was selected (voice or text), and the index of the cell in the `SDLChoiceSet` that was passed.

OBJC | SWIFT

PRESENTING THE MENU WITH A MODE

Finally, you will present the menu. When you do so, you must choose a `mode` to present it in. If you have no `vrCommands` on the choice cell you should choose `manualOnly`. If

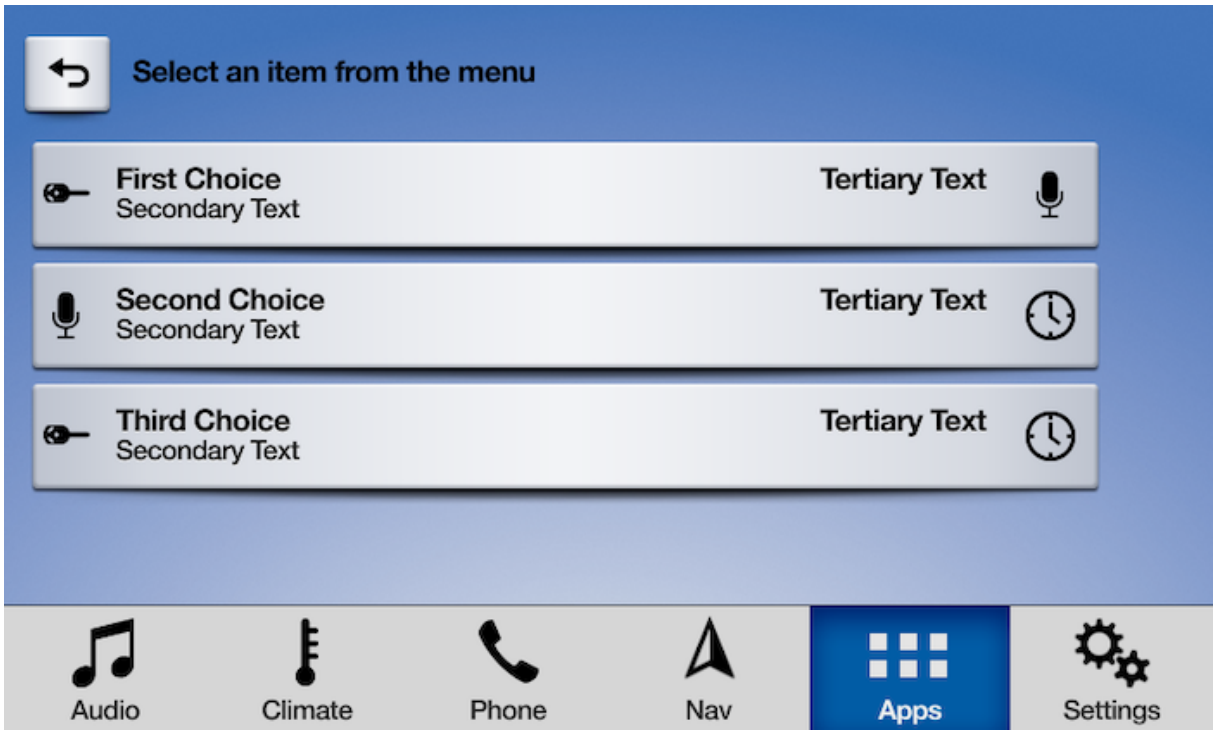
`vrCommands` are available, you may choose `voiceRecognitionOnly` or `both`.

You may want to choose this based on the trigger source leading to the menu being presented. For example, if the menu was presented via the user touching the screen, you may want to use a `mode` of `manualOnly` or `both`, but if the menu was presented via the user speaking a voice command, you may want to use a `mode` of `voiceRecognitionOnly` or `both`.

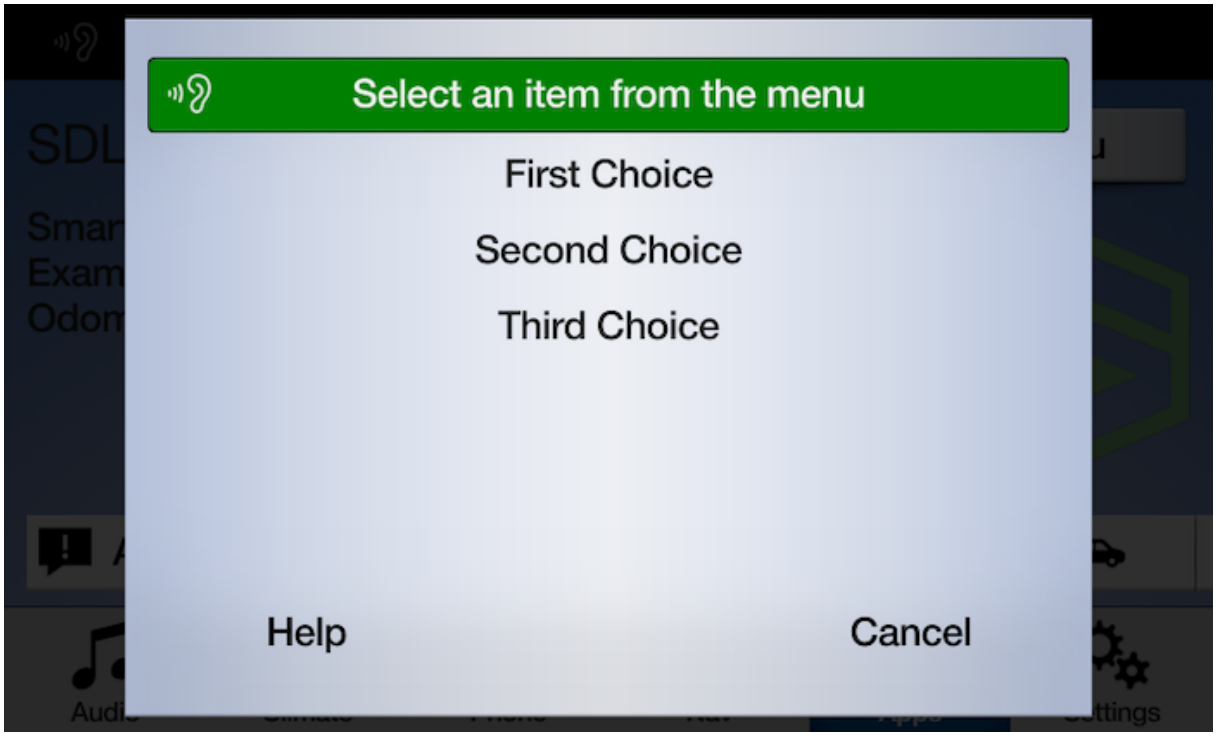
It may seem that the answer is to always use `both`. However, remember that you must provide `vrCommand`s on all cells to use `both`, which is exponentially slower than not providing `vrCommand`s (this is especially relevant for large menus, but less important for smaller ones). Also, some head units may not provide a good user experience for `both`.

INTERACTION MODE	DESCRIPTION
Manual only	Interactions occur only through the display
VR only	Interactions occur only through text-to-speech and voice recognition
Both	Interactions can occur both manually or through VR

MENU - MANUAL ONLY MODE



MENU - VOICE ONLY MODE

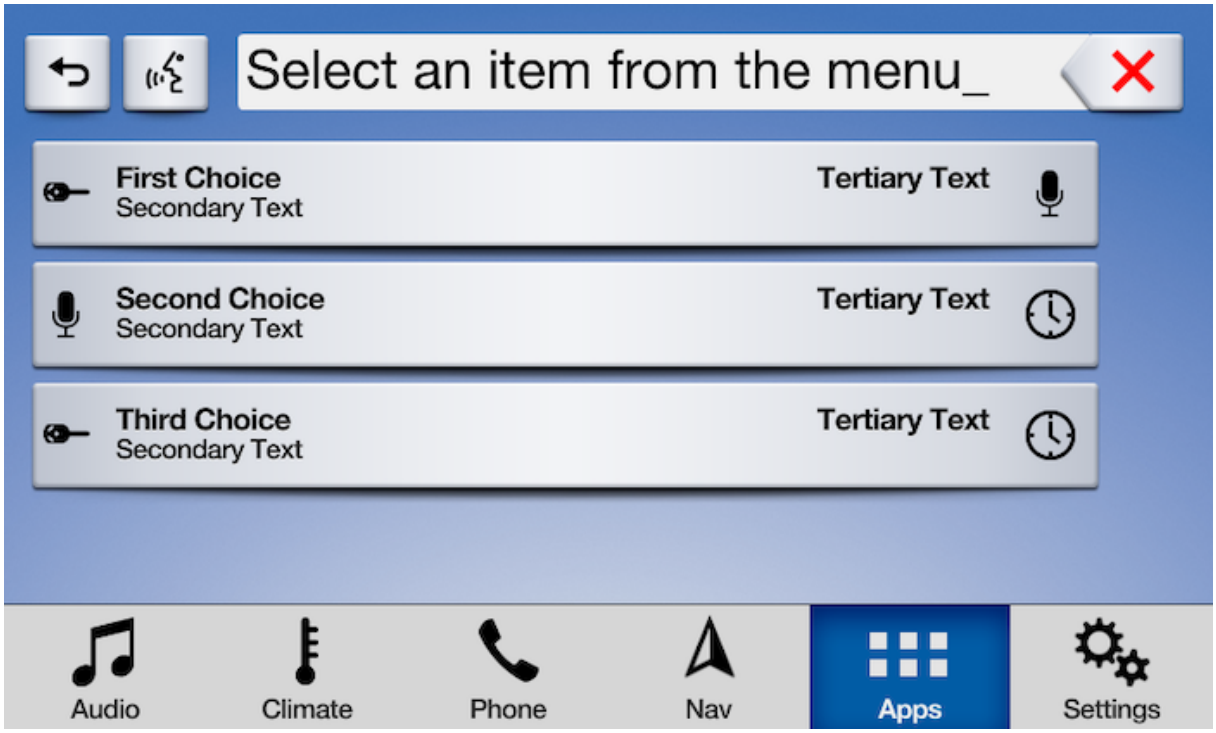


OBJC | SWIFT

Presenting a Searchable Menu

In addition to presenting a standard menu, you can also present a "searchable" menu, that is, a menu with a keyboard input box at the top. For more information on implementing the keyboard callbacks, see the Popup Keyboards guide.

MENU WITH SEARCH



OBJC | SWIFT

Deleting Cells

You can discover cells that have been preloaded on `screenManager.preloadedCells`. You may then pass an array of cells to delete from the remote system. Many times this is not necessary, but if you have deleted artwork used by cells, for example, you should delete the cells as well.

OBJC | SWIFT

Dismissing the Popup Menu (RPC v6.0+)

You can dismiss a displayed choice set before the timeout has elapsed by sending a `CancelInteraction` request. If you presented the choice set using the screen manager, you can dismiss the choice set by calling `cancel` on the `SDLChoiceCell` object that you presented.

> **NOTE**
>
> If connected to older head units that do not support this feature, the cancel request will be ignored, and the choice set will persist on the screen until the timeout has elapsed or the user dismisses it by making a selection.

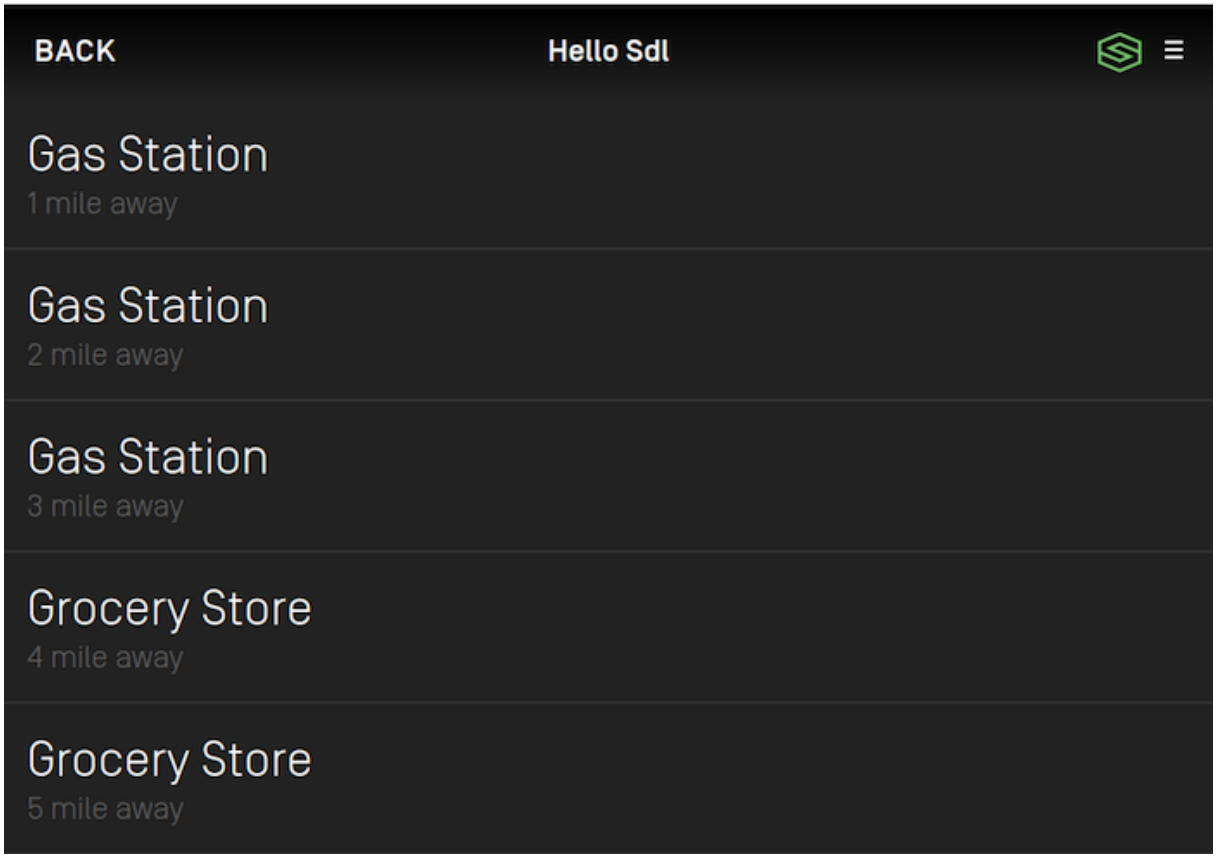
OBJC | SWIFT

Duplicate Cell Titles

Starting with SDL v7.1+ choice cells no longer require unique titles in order to be presented. For example, if you are trying to display points of interest as a list you can now have multiple locations with the same name but are not the same location. You cannot present multiple cells that are exactly the same. They must have some property that makes them different, such as `secondaryText` or an artwork.

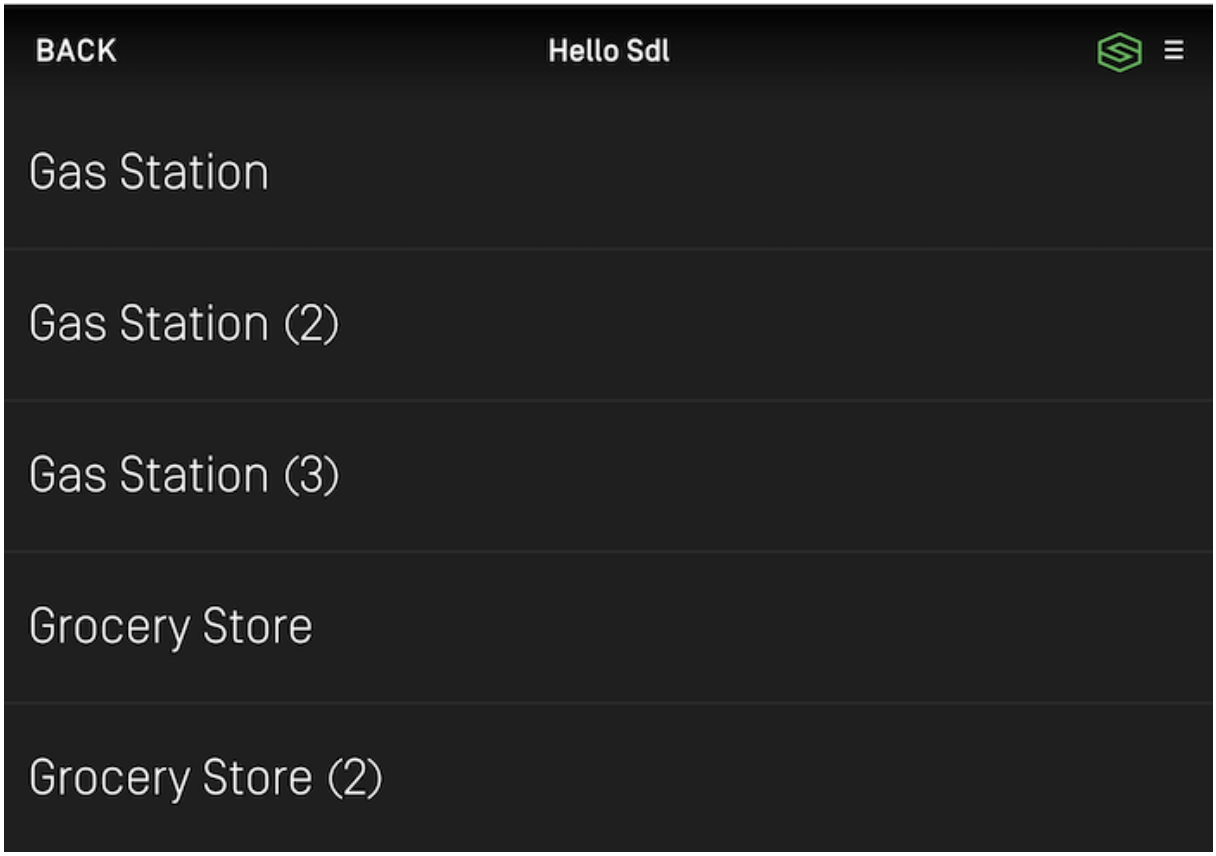
RPC V7.1+ CONNECTIONS

The titles on the choice set will be displayed as provided even if there are duplicate titles.



RPC V7.0 AND BELOW CONNECTIONS

The titles on the choice set will have a number appended to them when there are duplicate titles.



Using RPCs

If you don't want to use the `SDLScreenManager`, you can do this manually using the `Choice`, `CreateInteractionChoiceSet`, and `PerformInteraction`. You will need to create `Choice`s, bundle them into `CreateInteractionChoiceSet`s. As this is no longer a recommended course of action, we will leave it to you to figure out how to manually do it.

Note that if you do manually create a `PerformInteraction` and want to set a cancel id, the `SDLScreenManager` takes cancel ids 0 - 10000. Any cancel id you set must be outside of that range.

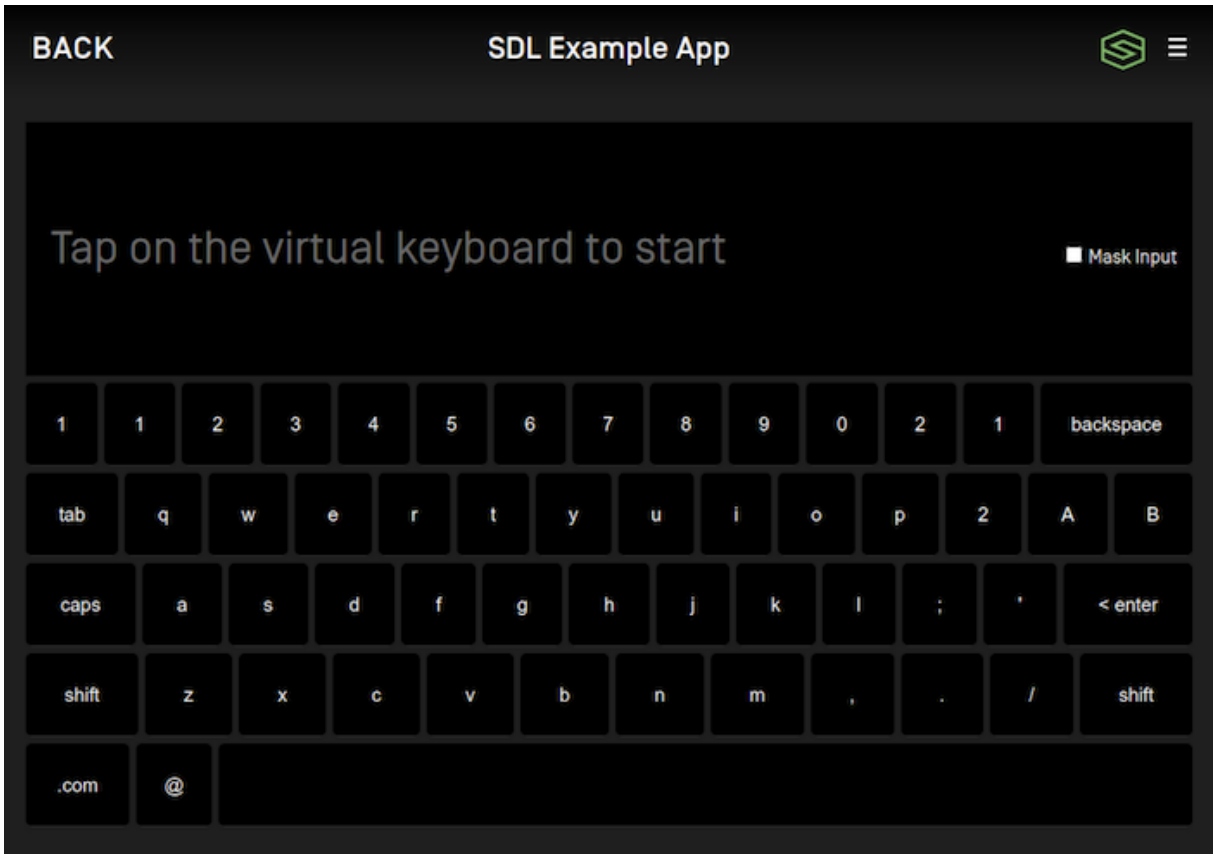
Popup Keyboards

Presenting a keyboard or a popup menu with a search field requires you to implement the `SDLKeyboardDelegate`. Note that the `initialText` in the keyboard case often acts as "placeholder text" and not as true initial text.

Presenting a Keyboard

You should present a keyboard to users when your app contains a "search" field. For example, in a music player app, you may want to give the user a way to search for a song or album. A keyboard could also be useful in an app that displays nearby points of interest, or in other situations.

> **NOTE**
>
> Keyboards are unavailable for use in many countries when the driver is distracted. This is often when the vehicle is moving above a certain speed, such as 5 miles per hour. This will be automatically managed by the system. Your keyboard may be disabled or an error returned if the driver is distracted.



OBJC | SWIFT

Implementing the Keyboard Delegate

Using the `SDLKeyboardDelegate` is required for popup keyboards and popup menus with search. It involves two required methods (for handling the user's input and the keyboard's unexpected abort), as well as several optional methods for additional functionality.

OBJC | SWIFT

Configuring Keyboard Properties

You can change default keyboard properties by updating `sdlManager.screenManager.keyboardConfiguration`. If you want to change the keyboard configuration for only one keyboard session and keep the default keyboard configuration unchanged, you can implement the `customKeyboardConfiguration` delegate method and pass back the single-use `KeyboardProperties` for that given keyboard presentation.

KEYBOARD LANGUAGE

You can modify the keyboard language by changing the keyboard configuration's `language`. For example, you can set an `EN_US` keyboard. It will default to `EN_US` if not otherwise set.

OBJC | SWIFT

LIMITED CHARACTER LIST

You can modify the keyboard to enable only some characters by responding to the `updateCharacterSet:completionHandler:` delegate method or by changing the keyboard configuration before displaying the keyboard. For example, you can enable only "a", "b" , and "c" on the keyboard. All other characters will be greyed out (disabled).

OBJC | SWIFT

AUTOCOMPLETE LIST

You can modify the keyboard to allow an app to pre-populate the text field with a list of suggested entries as the user types by responding to the `updateAutocompleteWithInput:autoCompleteResultsHandler:` delegate method or by changing the keyboard configuration before displaying the keyboard. For example, you can display recommended searches "test1", "test2", and "test3" if the user types "tes".

NOTE

A list of autocomplete results is only available on RPC 6.0+ connections. On connections < RPC 6.0, only the first item will be available to the user.

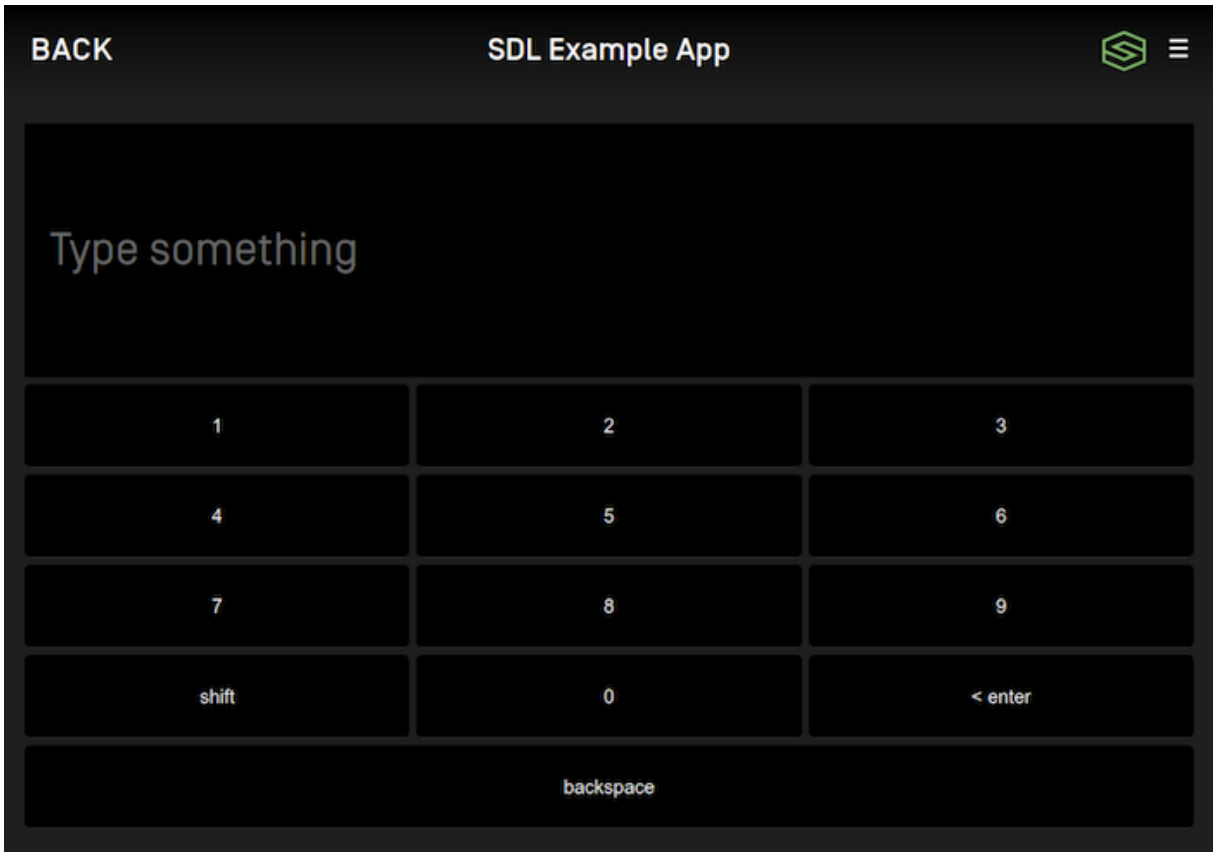
OBJC | SWIFT

KEYBOARD LAYOUT

You can modify the keyboard layout by changing the keyboard configuration's `keyboardLayout`. For example, you can set a `NUMERIC` keyboard. It will default to `QWERTY` if not otherwise set.

NOTE

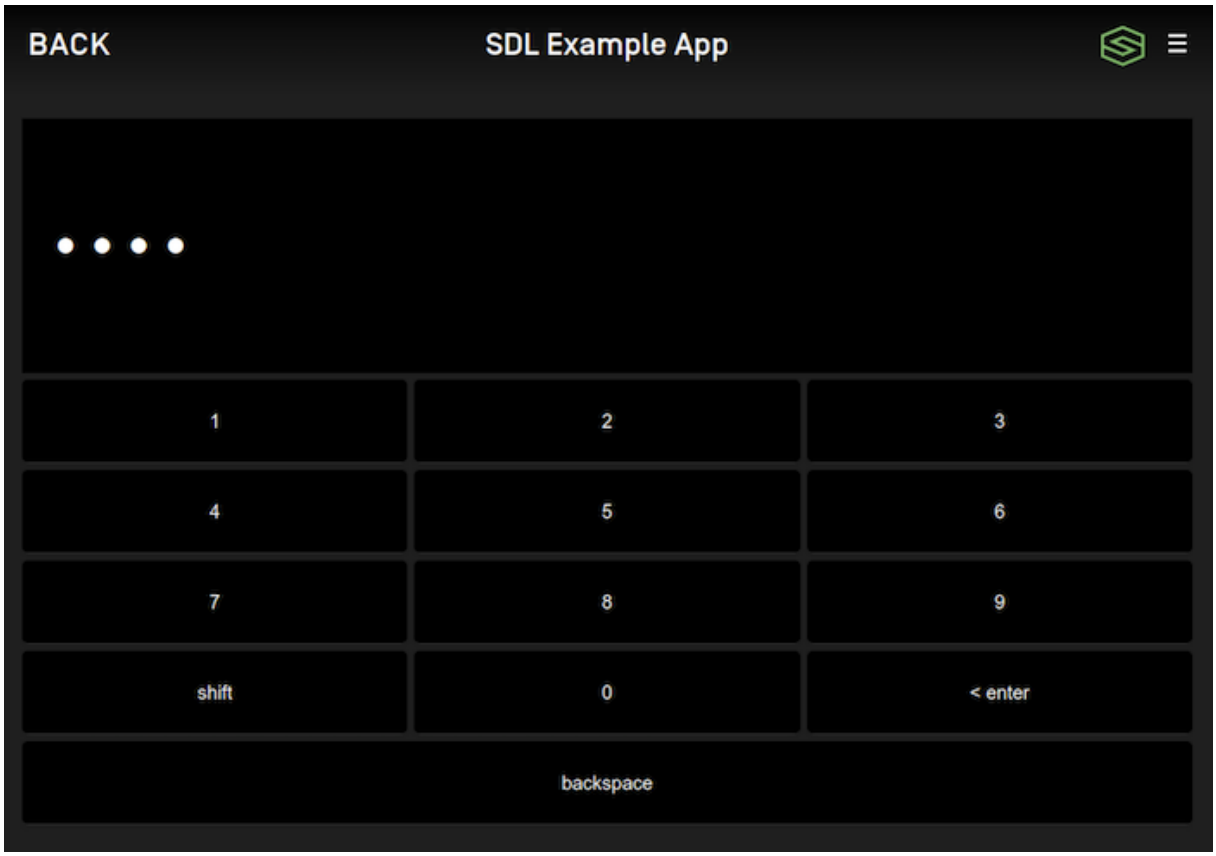
The numeric keyboard layout is only available on RPC 7.1+. See the section Checking Keyboard Capabilities to determine if this layout is available.



OBJC | SWIFT

INPUT MASKING (RPC 7.1+)

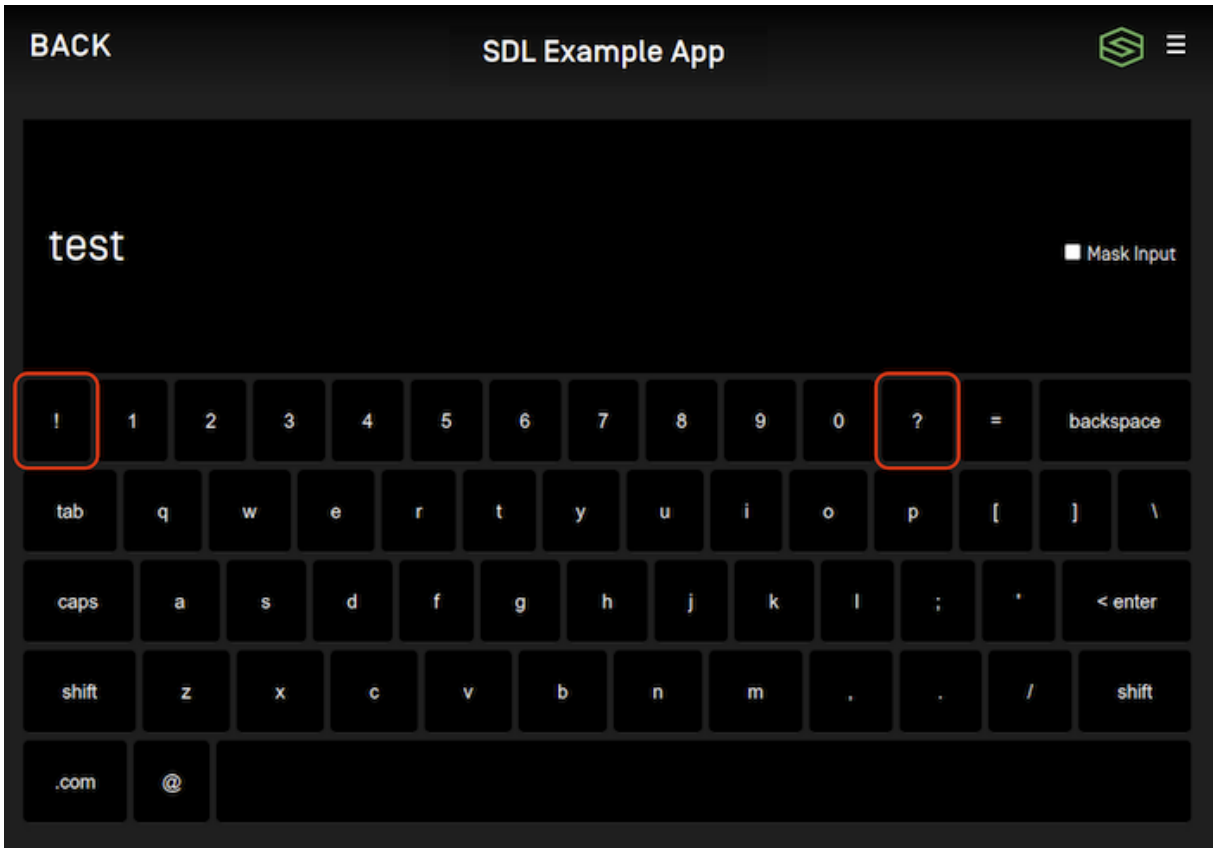
You can modify the keyboard to mask the entered characters by changing the keyboard configuration's `maskInputCharacters`.



OBJC | SWIFT

CUSTOM KEYS (RPC 7.1+)

Each keyboard layout has a number of keys that can be customized to your app's needs. For example, you could set two of the customizable keys in `QWERTY` layout to be "!" and "?" as seen in the image below. The available number and location of these custom keys is determined by the connected head unit. See the section Checking Keyboard Capabilities to determine how many custom keys are available for any given layout.



OBJC | SWIFT

Checking Keyboard Capabilities (RPC v7.1+)

Each head unit may support different keyboard layouts and each layout can support a different number of custom keys. Head units may not support masking input. If you want to know which keyboard features are supported on the connected head unit, you can check the `KeyboardCapabilities`:

OBJC | SWIFT

Dismissing the Keyboard (RPC v6.0+)

You can dismiss a displayed keyboard before the timeout has elapsed by sending a `CancelInteraction` request. If you presented the keyboard using the screen manager, you can dismiss the choice set by calling `dismissKeyboard` with the `cancelID` that was returned (if one was returned) when presenting.

> **NOTE**
>
> If connected to older head units that do not support this feature, the cancel request will be ignored, and the keyboard will persist on the screen until the timeout has elapsed or the user dismisses it by making a selection.

OBJC | SWIFT

Using RPCs

If you don't want to use the `SDLScreenManager`, you can do this manually using the `PerformInteraction` RPC request. As this is no longer a recommended course of action, we will leave it to you to figure out how to manually do it.

Note that if you do manually create a `PerformInteraction` and want to set a cancel id, the `SDLScreenManager` takes cancel ids 0 - 10000. Any cancel id you set must be outside of that range.

Alerts and Subtle Alerts

SDL supports two types of alerts: a large popup alert that typically takes over the whole screen and a smaller subtle alert that only covers a small part of screen.

Checking if the Module Supports Alerts

Your SDL app may be restricted to only being allowed to send an alert when your app is open (i.e. the `hmiLevel` is non-`NONE`) or when it is the currently active app (i.e. the `hmiLevel` is `FULL`). Subtle alert is a new feature (RPC v7.0+) and may not be supported on all modules.

OBJC | SWIFT

Alerts

An alert is a large pop-up window showing a short message with optional buttons. When an alert is activated, it will abort any SDL operation that is in-progress, except the already-in-progress alert. If an alert is issued while another alert is still in progress the newest alert will wait until the current alert has finished.

Depending on the platform, an alert can have up to three lines of text, a progress indicator (e.g. a spinning wheel or hourglass), and up to four soft buttons.

ALERT WITH NO SOFT BUTTONS

> **NOTE**
>
> If no soft buttons are added to an alert some modules may add a default "cancel" or "close" button.

ALERT WITH SOFT BUTTONS

Creating the AlertView

Use the `SDLAlertView` to set all the properties of the alert you want to present.

> **NOTE**
>
> An `SDLAlertView` must contain at least either `text`, `secondaryText` or `audio` for the alert to be presented.

TEXT

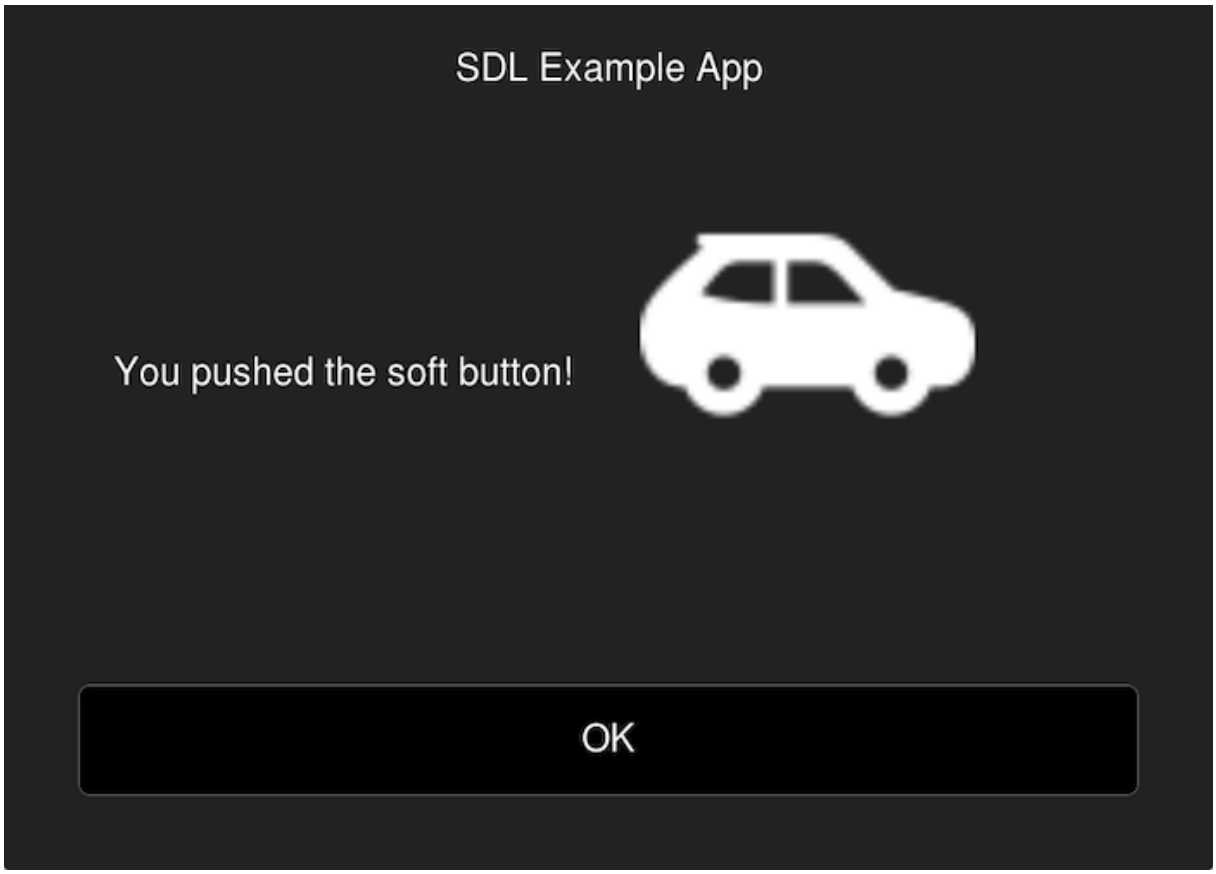
OBJC | SWIFT

BUTTONS

OBJC | SWIFT

ICON

An alert can include a custom or static (built-in) image that will be displayed within the alert.



OBJC | SWIFT

TIMEOUTS

An optional timeout can be added that will dismiss the alert when the duration is over. Typical timeouts are between 3 and 10 seconds. If omitted, a default of 5 seconds is used.

OBJC | SWIFT

PROGRESS INDICATOR

Not all modules support a progress indicator. If supported, the alert will show an animation that indicates that the user must wait (e.g. a spinning wheel or hourglass, etc).

If omitted, no progress indicator will be shown.

OBJECTIVE-C

OBJC | SWIFT

TEXT-TO-SPEECH

An alert can also speak a prompt or play a sound file when the alert appears on the screen. This is done by creating an `SDLAlertAudioData` object and setting it in the `SDLAlertView`

> **NOTE**
>
> On Manticore, using alerts with audio (Text-To-Speech or Tones) work best in Google Chrome, Mozilla Firefox, or Microsoft Edge. Alerts with audio does not work in Apple Safari at this time.

OBJC | SWIFT

`SDLAlertAudioData` can also play an audio file.

OBJC | SWIFT

You can also play a combination of audio files and text-to-speech strings. The audio will be played in the order you add them to the `SDLAlertAudioData` object.

OBJC | SWIFT

PLAY TONE

To play a notification sound when the alert appears, set `playTone` to `true`.

OBJC | SWIFT

Showing the Alert

OBJC | SWIFT

Canceling/Dismissing the Alert

You can cancel an alert that has not yet been sent to the head unit.

On systems with RPC v6.0+ you can dismiss a displayed alert before the timeout has elapsed. This feature is useful if you want to show users a loading screen while performing a task, such as searching for a list for nearby coffee shops. As soon as you have the search results, you can cancel the alert and show the results.

NOTE

If connected to older head units that do not support this feature, the cancel request will be ignored, and the alert will persist on the screen until the timeout has elapsed or the user dismisses the alert by selecting a button.

> **NOTE**
>
> Canceling the alert will only dismiss the displayed alert. If the alert has audio, the speech will play in its entirety even when the displayed alert has been dismissed. If you know you will cancel an alert, consider setting a short audio message like "searching" instead of "searching for coffee shops, please wait."

OBJC | SWIFT

Using RPCs

You can also use RPCs to present alerts. You need to use the `Alert` RPC to do so. Note that if you do so, you must avoid using soft button ids 0 - 10000 and cancel ids 0 - 10000 because these ranges are used by the `ScreenManager`.

Subtle Alerts (RPC v7.0+)

A subtle alert is a notification style alert window showing a short message with optional buttons. When a subtle alert is activated, it will not abort other SDL operations that are in-progress like the larger pop-up alert does. If a subtle alert is issued while another subtle alert is still in progress the newest subtle alert will simply be ignored.

Touching anywhere on the screen when a subtle alert is showing will dismiss the alert. If the SDL app presenting the alert is not currently the active app, touching inside the subtle alert will open the app.

Depending on the platform, a subtle alert can have up to two lines of text and up to two soft buttons.

NOTE

Because `SubtleAlert` is not currently supported in the `ScreenManager`, you need to be careful when setting soft buttons or cancel ids to ensure that they do not conflict with those used by the `ScreenManager`. The `ScreenManager` takes soft button ids 0 - 10000 and cancel ids 0 - 10000. Ensure that if you use custom RPCs that the soft button ids and cancel ids are outside of this range.

SUBTLE ALERT WITH NO SOFT BUTTONS

SUBTLE ALERT WITH SOFT BUTTONS

Creating the Subtle Alert

The following steps show you how to add text, images, buttons, and sound to your subtle alert. Please note that at least one line of text or the "text-to-speech" chunks must be set in order for your subtle alert to work.

TEXT

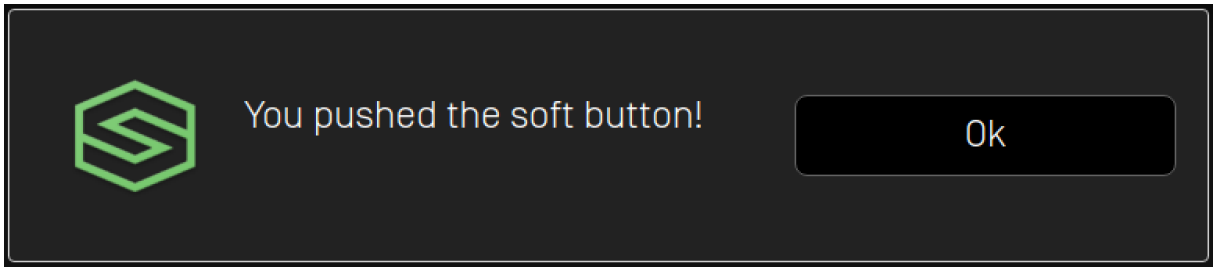
OBJC | SWIFT

BUTTONS

OBJC | SWIFT

ICON

A subtle alert can include a custom or static (built-in) image that will be displayed within the subtle alert. Before you add the image to the subtle alert, make sure the image is uploaded to the head unit using the `SDLFileManager`. Once the image is uploaded, you can show the alert with the icon.



OBJC | SWIFT

TIMEOUTS

An optional timeout can be added that will dismiss the subtle alert when the duration is over. Typical timeouts are between 3 and 10 seconds. If omitted, a default of 5 seconds is

used.

OBJC SWIFT

```
// Duration timeout is in milliseconds
subtleAlert.duration = @4000;
```

TEXT-TO-SPEECH

A subtle alert can also speak a prompt or play a sound file when the subtle alert appears on the screen. This is done by setting the `ttsChunks` parameter.

OBJC SWIFT

```
subtleAlert.ttsChunks = [SDLTTSChunk textChunksFromString:<#(nonnull NSString
*)#>];
```

The `ttsChunks` parameter can also take a file to play/speak. For more information on how to upload the file please refer to the Playing Audio Indications guide.

OBJC SWIFT

```
subtleAlert.ttsChunks = [SDLTTSChunk fileChunksWithName:<#(nonnull NSString
*)#>];
```

Showing the Subtle Alert

OBJC SWIFT

```
[self.sdlManager sendRequest:subtleAlert withResponseHandler:^(SDLRPCRequest
*request, SDLRPCResponse *response, NSError *error) {
    if (!response.success.boolValue) {
        <#Print out the error if there is one#>
        return;
    }

    <#Subtle alert was shown successfully#>
}];
```

Checking if the User Dismissed the Subtle Alert

If desired, you can be notified when the user tapped on the subtle alert by registering for the `SDLOnSubtleAlertPressed` notification.

OBJC SWIFT

```
[self.sdlManager subscribeToRPC:SDLDidReceiveSubtleAlertPressedNotification
withObserver:self selector:@selector(subtleAlertPressed)];

- (void)subtleAlertPressed {
    <#The subtle alert was pressed#>
}
```

Dismissing the Subtle Alert

You can dismiss a displayed subtle alert before the timeout has elapsed.

> **NOTE**
>
> Canceling the subtle alert will only dismiss the displayed alert. If you have set the `ttsChunk` property, the speech will play in its entirety even when the displayed subtle alert has been dismissed. If you know you will cancel a subtle alert, consider setting a short `ttsChunk`.

There are two ways to dismiss a subtle alert. The first way is to dismiss a specific subtle alert using a unique `cancelID` assigned to the subtle alert. The second way is to dismiss whichever subtle alert is currently on-screen.

DISMISSING A SPECIFIC SUBTLE ALERT

OBJC SWIFT

```
// `cancelID` is the ID that you assigned when creating and sending the subtle alert
SDLCancelInteraction *cancelInteraction = [[SDLCancelInteraction alloc]
initWithSubtleAlertCancelID:cancelID];
[self.sdlManager sendRequest:cancelInteraction withResponseHandler:^(__kindof
SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse * _Nullable
response, NSError * _Nullable error) {
    if (!response.success.boolValue) {
        <#Print out the error if there is one#>
        return;
    }
    <#The subtle alert was canceled successfully#>
}];
```

DISMISSING THE CURRENT SUBTLE ALERT

OBJC | SWIFT

```
SDLCancelInteraction *cancelInteraction = [SDLCancelInteraction subtleAlert];
[self.sdlManager sendRequest:cancelInteraction withResponseHandler:^(__kindof
SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse * _Nullable
response, NSError * _Nullable error) {
    if (!response.success.boolValue) {
        <#Print out the error if there is one#>
        return;
    }
    <#The subtle alert was canceled successfully#>
}];
```

Media Clock

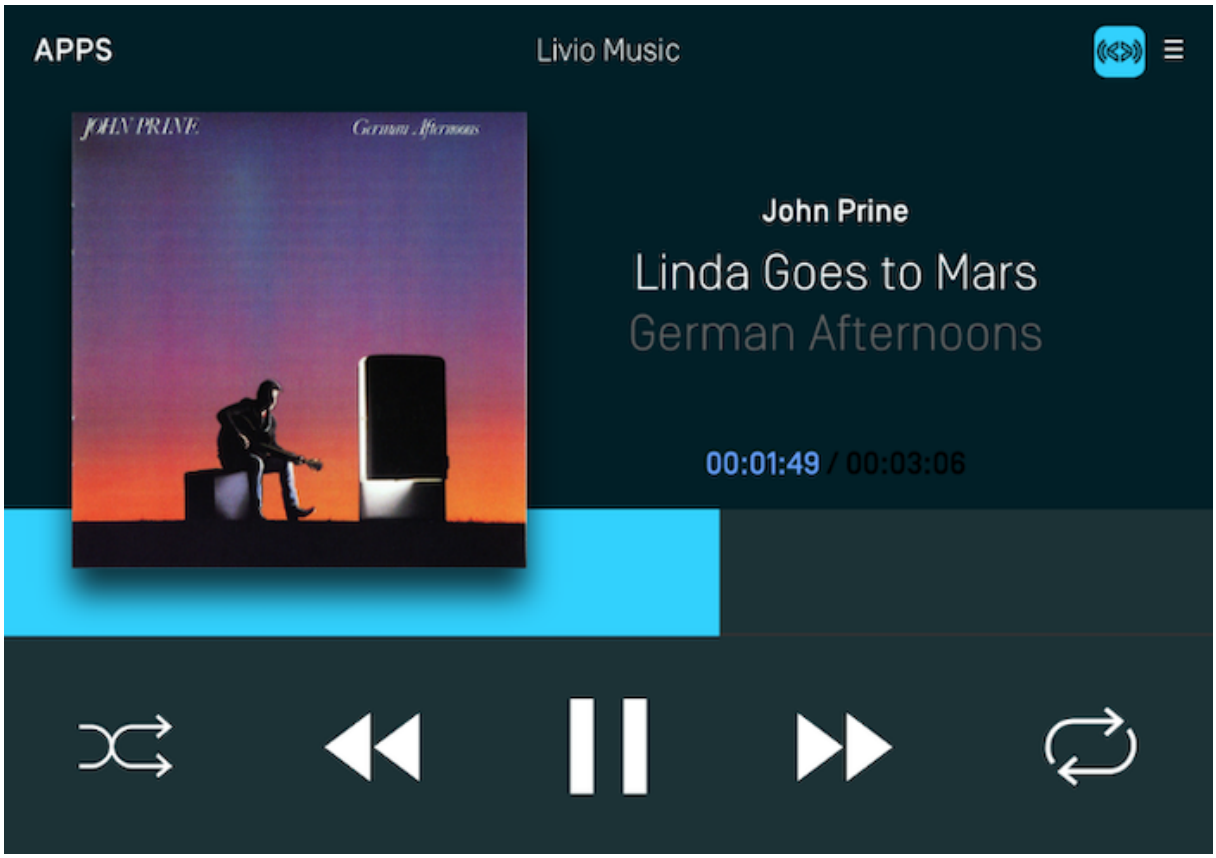
The media clock is used by media apps to present the current timing information of a playing media item such as a song, podcast, or audiobook.

The media clock consists of three parts: the progress bar, a current position label and a remaining time label. In addition, you may want to update the play/pause button icon to

reflect the current state of the audio or the media forward / back buttons to reflect if it will skip tracks or time.

> **NOTE**
>
> Media clock operations require the HMI status to be `FULL`. More information on how to monitor the HMI status can be found in the Understanding Permissions guide.

> **NOTE**
>
> Ensure your app has an `appType` of media and you are using the media template before implementing this feature.



Counting Up

In order to count up using the timer, you will need to set a start time that is less than the end time. The "bottom end" of the media clock will always start at `0:00` and the "top end" will be the end time you specified. The start time can be set to any position between 0 and the end time. For example, if you are starting a song at `0:30` and it ends at `4:13` the media clock timer progress bar will start at the `0:30` position and start incrementing up automatically every second until it reaches `4:13`. The current position label will start counting upwards from `0:30` and the remaining time label will start counting down from `3:43`. When the end is reached, the current time label will read `4:13`, the remaining time label will read `0:00` and the progress bar will stop moving.

The play / pause indicator parameter is used to update the play / pause button to your desired button type. This is explained below in the section "Updating the Audio Indicator"

OBJC | SWIFT

Counting Down

Counting down is the opposite of counting up (I know, right?). In order to count down using the timer, you will need to set a start time that is greater than the end time. The timer bar moves from right to left and the timer will automatically count down. For example, if you're counting down from `10:00` to `0:00`, the progress bar will be at the leftmost position and start decrementing every second until it reaches `0:00`.

OBJC | SWIFT

Pausing & Resuming

When pausing the timer, it will stop the timer as soon as the request is received and processed. When a resume request is sent, the timer begins again at the paused time as soon as the request is processed. You can update the start and end times using a pause command to change the timer while remaining paused.

OBJC | SWIFT

Clearing the Timer

Clearing the timer removes it from the screen.

OBJC | SWIFT

Setting the Play / Pause Button Style (RPC v5.0+)

The audio indicator is, essentially, the play / pause button. You can tell the system which icon to display on the play / pause button to correspond with how your app works. For example, if audio is currently playing you can update the play/pause button to show the pause icon. On older head units, the audio indicator shows an icon with both the play and pause indicators and the icon can not be updated.

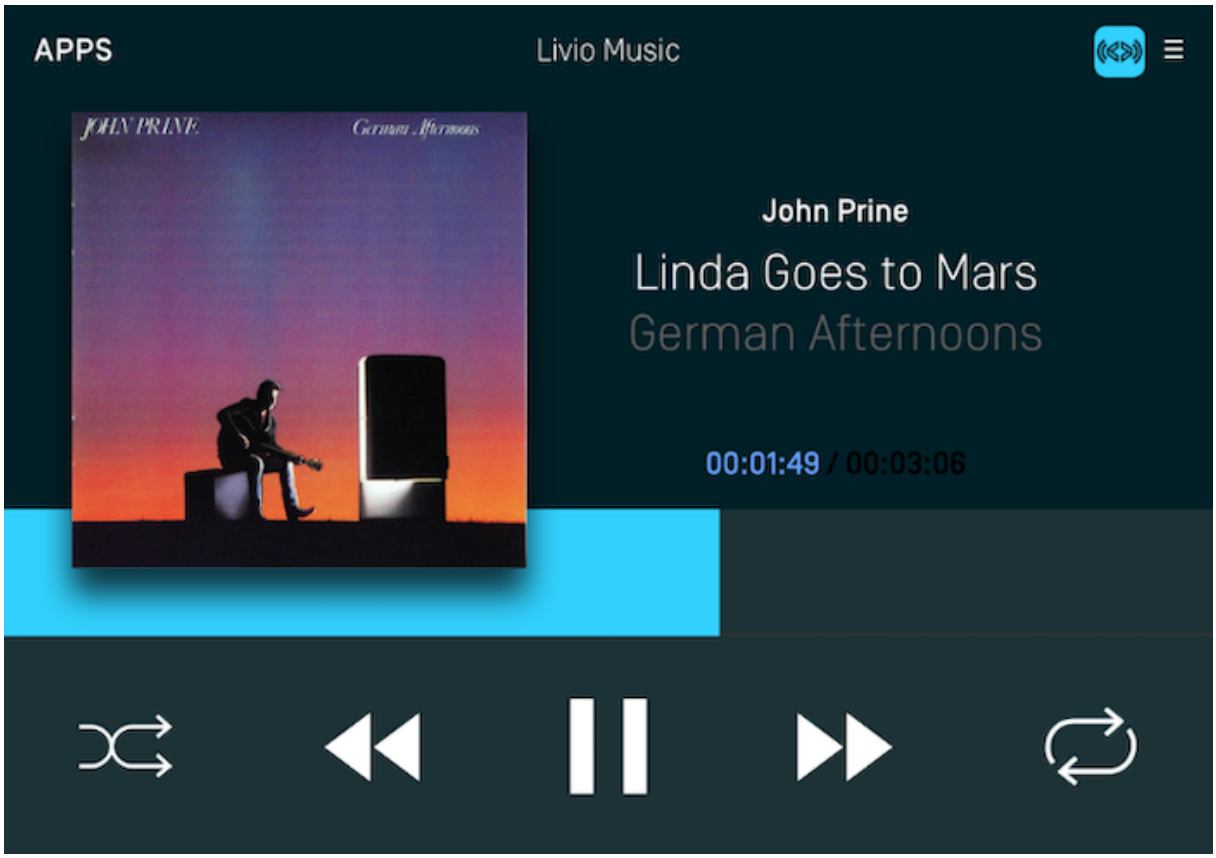
For example, a radio app will probably want two button states: play and stop. A music app, in contrast, will probably want a play and pause button. If you don't send any audio indicator information, a play / pause button will be displayed.

Setting The Media Forward / Back Button Style (RPC v7.1+)

As of RPC v7.1, you can set the style of the media forward / back buttons to show icons for skipping time (in seconds) forward and backward instead of skipping tracks. The skipping time style is common in podcast & audiobook media apps.

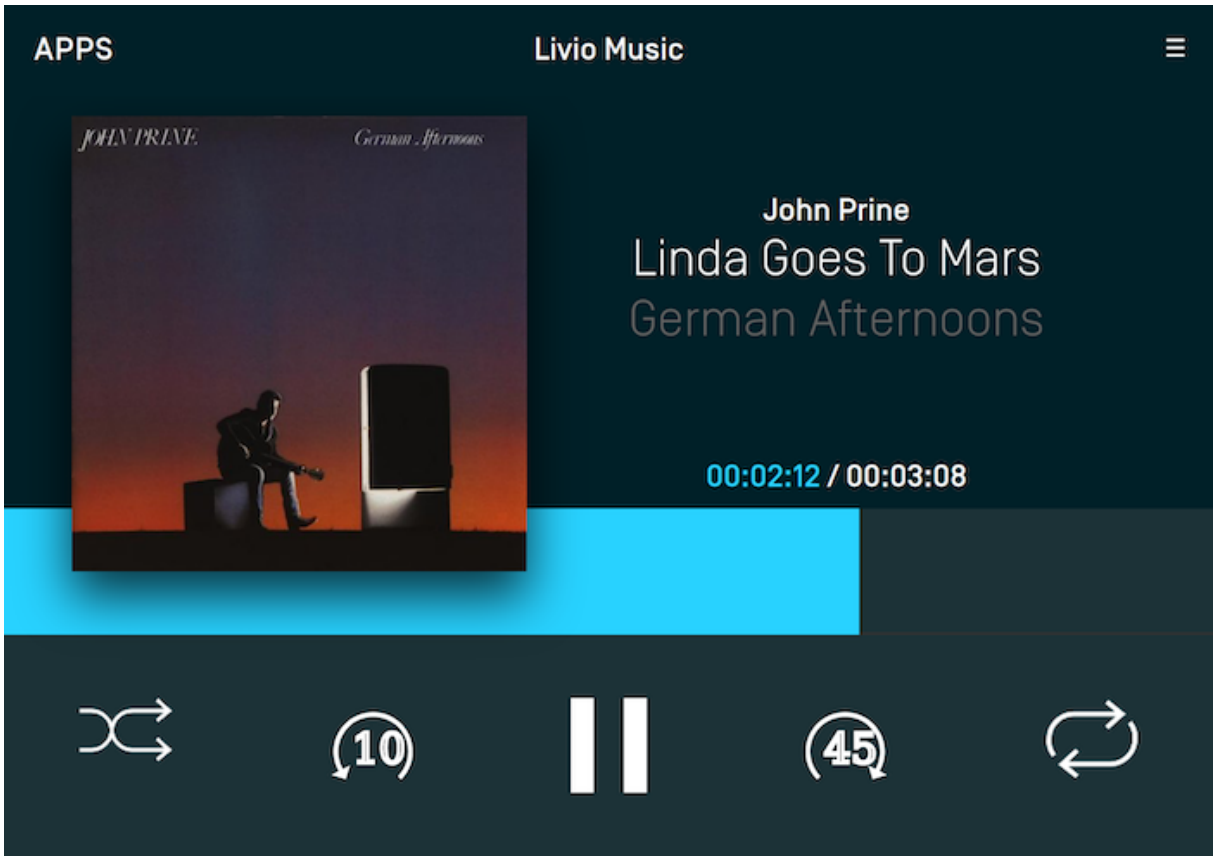
When you set the skip indicator style, you can set type `TRACK`, which is the default style that shows "skip forward" and "skip back" indicators. This is the only style available on RPC < 7.1 connections. You can also set the new type `TIME`, which will allow you to set the number of seconds and display indicators for skipping forward and backward in time.

Track Style



OBJC | SWIFT

Time Style



OBJC | SWIFT

Adding Custom Playback Rate (RPC v7.1+)

Many audio apps that support podcasts and audiobooks allow the user to adjust the audio playback rate.
As of RPC v7.1, you can set the rate that the audio is playing at to ensure the media clock accurately reflects the audio.

For example, a user can play a podcast at 125% speed or at 75% speed.

OBJC | SWIFT

NOTE

`CountRate` has a default value of 1.0, and the `CountRate` will be reset to 1.0 if any `SetMediaClockTimer` request does not have the parameter set. To ensure that you maintain the correct `CountRate` in your application make sure to set the parameter in all `SetMediaClockTimer` requests (including when sending a RESUME request).

Slider

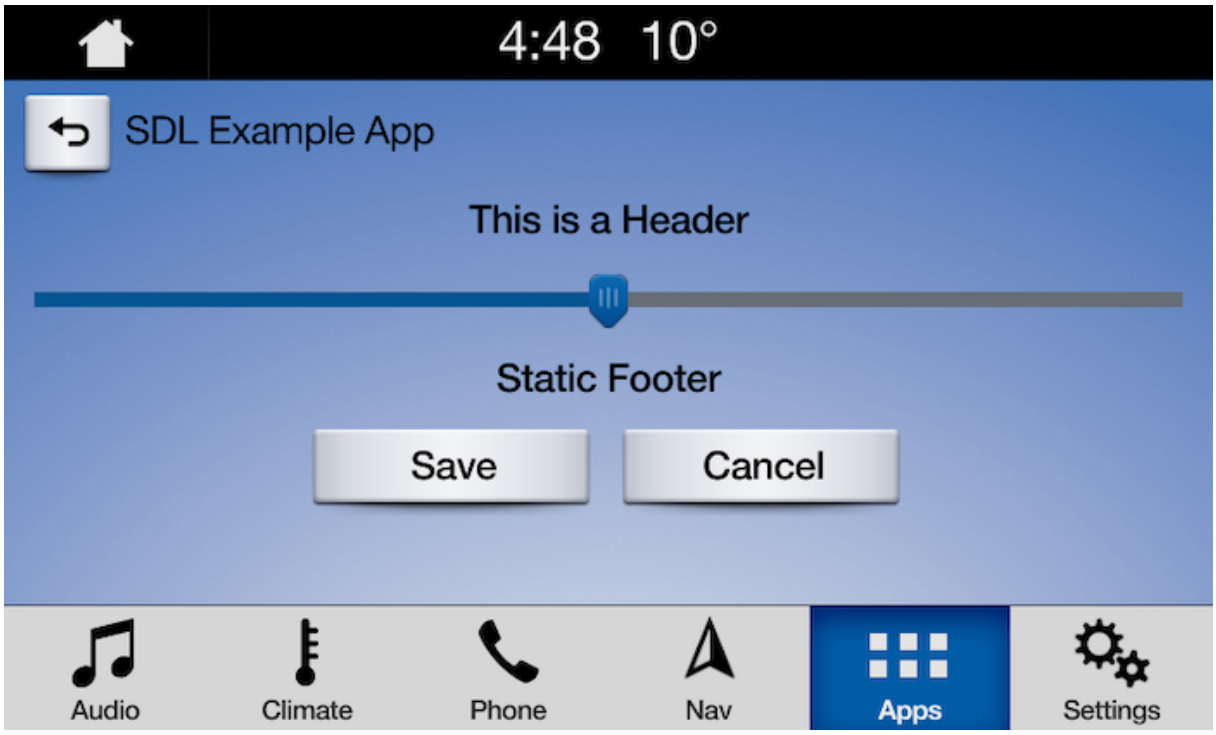
A `SDLSlider` creates a full screen or pop-up overlay (depending on platform) that a user can control. There are two main `SDLSlider` layouts, one with a static footer and one with a dynamic footer.

NOTE

The slider will persist on the screen until the timeout has elapsed or the user dismisses the slider by selecting a position or canceling.

A slider popup with a static footer displays a single, optional, footer message below the slider UI. A dynamic footer can show a different message for each slider position.

Slider UI



DYNAMIC SLIDER IN POSITION 1

DYNAMIC SLIDER IN POSITION 2

Creating the Slider

OBJC | SWIFT

Ticks

The number of selectable items on a horizontal axis.

OBJC | SWIFT

Position

The initial position of slider control (cannot exceed numTicks).

OBJC | SWIFT

Header

The header to display.

OBJC | SWIFT

Static Footer

The footer will have the same message across all positions of the slider.

OBJC | SWIFT

Dynamic Footer

This type of footer will have a different message displayed for each position of the slider. The footer is an optional parameter. The footer message displayed will be based off of the slider's current position. The footer array should be the same length as `numTicks` because each footer must correspond to a tick value. Or, you can pass `nil` to have no footer at all.

OBJC | SWIFT

Cancel ID

An ID for this specific slider to allow cancellation through the `CancelInteraction` RPC. The `ScreenManager` takes cancel ids 0 - 10000, so ensure any cancel id that you set is outside of that range.

OBJC | SWIFT

Show the Slider

OBJC | SWIFT

Dismissing a Slider (RPC v6.0+)

You can dismiss a displayed slider before the timeout has elapsed by dismissing either a specific slider or the current slider.

> **NOTE**
>
> If connected to older head units that do not support this feature, the cancel request will be ignored, and the slider will persist on the screen until the timeout has elapsed or the user dismisses by selecting a position or canceling.

Dismissing a Specific Slider

OBJC | SWIFT

Dismissing the Current Slider

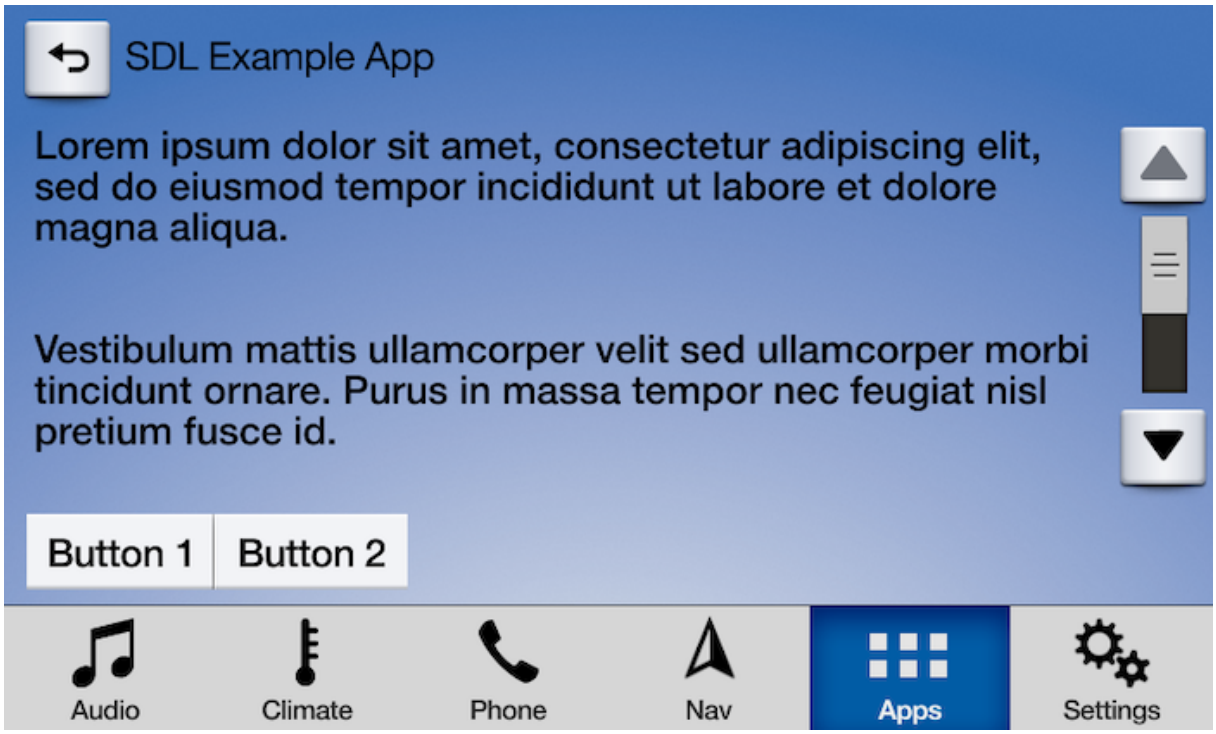
OBJC | SWIFT

Scrollable Message

A `SDLScrollableMessage` creates an overlay containing a large block of formatted text that can be scrolled. It contains a body of text, a message timeout, and up to eight soft buttons. To display a scrollable message in your SDL app, you simply send an `SDLScrollableMessage` RPC request.

> **NOTE**
>
> The message will persist on the screen until the timeout has elapsed or the user dismisses the message by selecting a soft button or cancelling (if the head unit provides cancel UI).

Scrollable Message UI



Creating the Scrollable Message

Currently, you can only create a scrollable message view to display on the screen using RPCs.

NOTE

The `SDLScreenManager` uses soft button ids 0 – 10000. Ensure that if you use custom RPCs—such as this one—that the soft button ids you use are outside of this range (i.e. > 10000).

OBJC | SWIFT

Dismissing a Scrollable Message (RPC v6.0+)

You can dismiss a displayed scrollable message before the timeout has elapsed. You can dismiss a specific scrollable message, or you can dismiss the scrollable message that is currently displayed.

NOTE

If connected to older head units that do not support this feature, the cancel request will be ignored, and the scrollable message will persist on the screen until the timeout has elapsed or the user dismisses the message by selecting a button.

Dismissing a Specific Scrollable Message

OBJC | SWIFT

Dismissing the Current Scrollable Message

OBJC | SWIFT

Customizing the Template

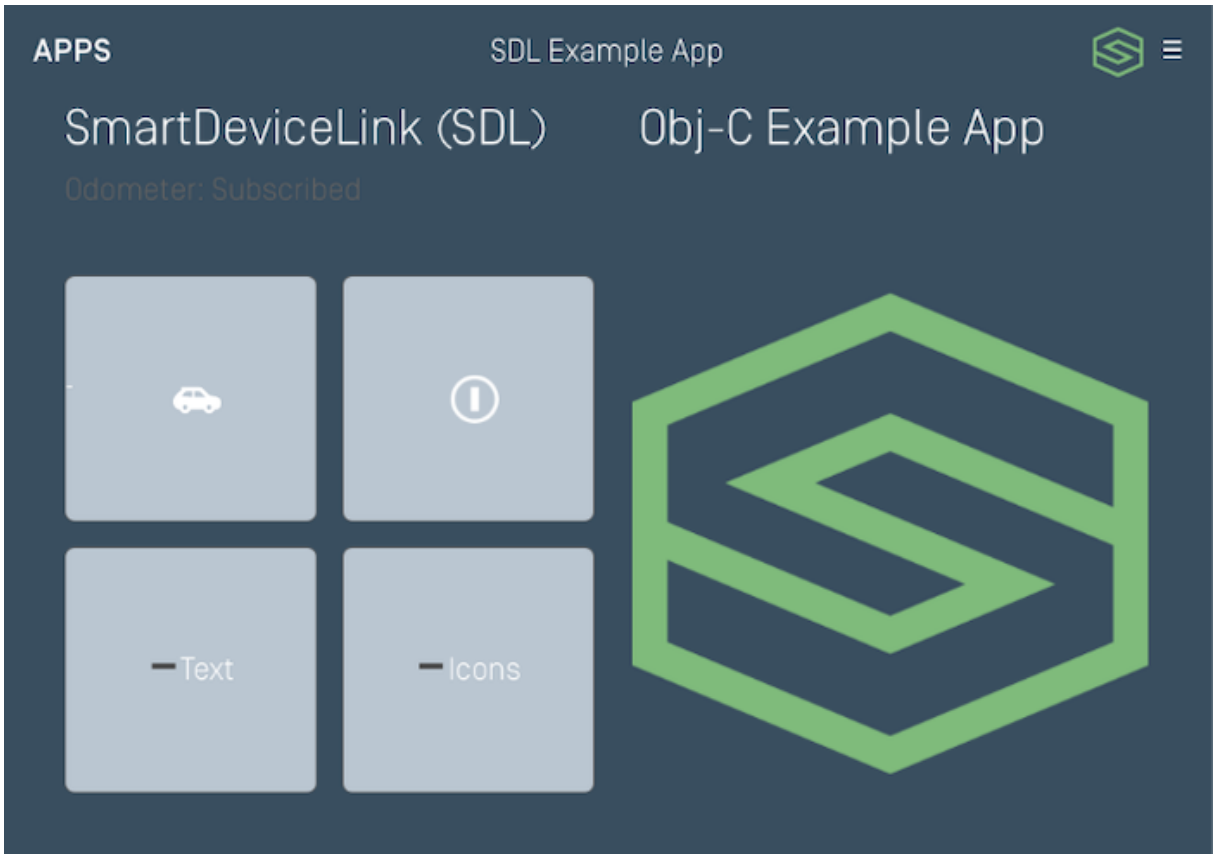
You have the ability to customize the look and feel of the template. How much customization is available depends on the RPC version of the head unit you are connected with as well as the design of the HMI.

Customizing Template Colors (RPC v5.0+)

You can customize the color scheme of your app using template coloring APIs.

Customizing the Default Layout

You can change the template colors of the initial template layout in the `lifecycleConfiguration`.



OBJC | SWIFT

NOTE

You may only change the template coloring once per template; that is, you cannot call `changeLayout`, `SetDisplayLayout` or `Show` for the template you are already on and expect the color scheme to update.

Customizing Future Layouts

You can change the template color scheme when you change layouts. This guide requires SDL iOS version 7.0. If using an older version, use `SDLSetDisplayLayout` (any RPC version) or `SDLShow` (RPC v6.0+) request.

OBJC | SWIFT

Customizing the Menu Title and Icon

You can also customize the title and icon of the main menu button that appears on your template layouts. The menu icon must first be uploaded with a specific name through the file manager; see the Uploading Images section for more information on how to upload your image.

OBJC | SWIFT

Customizing the Keyboard (RPC v3.0+)

If you present keyboards in your app – such as in searchable interactions or another custom keyboard – you may wish to customize the keyboard for your users. The best way to do this is through the `SDLScreenManager`. For more information presenting keyboards, see the Popup Keyboards section.

Setting Keyboard Properties

You can modify the language of the keyboard to change the characters that are displayed.

OBJC | SWIFT

Other Properties

While there are other keyboard properties available on `SDLKeyboardProperties`, these will be overridden by the screen manager. The `keypressMode` must be a specific configuration for the screen manager's callbacks to work properly. The `limitedCharacterList`, `autoCompleteText`, and `autoCompleteList` will be set on a per-keyboard basis in the `SDLKeyboardDelegate` which is set on the `presentKeyboard` and `presentSearchableChoiceSet` methods.

Customizing Help Prompts

On some head units it is possible to display a customized help menu or speak a custom command if the user asks for help while using your app. The help menu is commonly used to let users know what voice commands are available, however, it can also be customized to help your user navigate the app or let them know what features are available.

Configuring the Help Menu

You can customize the help menu with your own title and/or menu options. If you don't customize these options, then the head unit's default menu will be used.

If you wish to use an image, you should check the `sdlManager.systemCapabilityManager.defaultMainWindowCapability.imageFields` for an `imageField.name` of `vrHelpItem` to see if that image is supported. If `vrHelpItem` is in the `imageFields` array, then it can be used. You will then need to upload the image using the file manager before using it in the request. See the Uploading Images section for more information.

OBJC | SWIFT

Configuring the Help Prompt

On head units that support voice recognition, a user can request assistance by saying "Help." In addition to displaying the help menu discussed above a custom spoken text-to-

speech response can be spoken to the user.

OBJC | SWIFT

Configuring the Timeout Prompt

If you display any sort of popup menu or modal interaction that has a timeout – such as an alert, interaction, or slider – you can create a custom text-to-speech response that will be spoken to the user in the event that a timeout occurs.

OBJC | SWIFT

Clearing Help Menu and Prompt Customizations

You can also reset your customizations to the help menu or spoken prompts. To do so, you will send a `ResetGlobalProperties` RPC with the fields that you wish to clear.

OBJC | SWIFT

Playing Spoken Feedback

Since your user will be driving while interacting with your SDL app, speech phrases can provide important feedback to your user. At any time during your app's lifecycle you can send a speech phrase using the `SDLSpeak` request and the head unit's text-to-speech (TTS) engine will produce synthesized speech from your provided text.

When using the `SDLSpeak` RPC, you will receive a response from the head unit once the operation has completed. From the response you will be able to tell if the speech was completed, interrupted, rejected or aborted. It is important to keep in mind that a speech request can interrupt another ongoing speech request. If you want to chain speech requests you must wait for the current speech request to finish before sending the next speech request.

> **NOTE**
>
> On Manticore, spoken feedback works best in Google Chrome, Mozilla Firefox, or Microsoft Edge. Spoken feedback does not work in Apple Safari at this time.

Creating the Speak Request

The speech request you send can simply be a text phrase, which will be played back in accordance with the user's current language settings, or it can consist of phoneme specifications to direct SDL's TTS engine to speak a language-independent, speech-sculpted phrase. It is also possible to play a pre-recorded sound file (such as an MP3) using the speech request. For more information on how to play a sound file please refer to Playing Audio Indications.

Getting the Supported Speech Capabilities

Once you have successfully connected to the module, you can access supported speech capabilities properties on the `SDLManager.systemCapabilityManager` instance.

OBJC | SWIFT

Below is a list of commonly supported speech capabilities.

SPEECH CAPABILITY	DESCRIPTION
Text	Text phrases
SAPI Phonemes	Microsoft speech synthesis API
File	A pre-recorded sound file

Creating Different Types of Speak Requests

Once you know what speech capabilities are supported by the module, you can create the speak requests.

TEXT PHRASE

OBJC | SWIFT

SAPI PHONEMES PHRASE

OBJC | SWIFT

Sending the Speak Request

OBJC | SWIFT

Playing Audio Indications (RPC v5.0+)

You can pass an uploaded audio file's name to `SDLTTSChunk`, allowing any API that takes a text-to-speech parameter to pass and play your audio file. A sports app, for example, could play a distinctive audio chime to notify the user of a score update alongside an `Alert` request.

> **NOTE**
>
> On Manticore, audio indications work best in Google Chrome, Mozilla Firefox, or Microsoft Edge. Audio indications do not work in Apple Safari at this time.

Uploading the Audio File

The first step is to make sure the audio file is available on the remote system. To upload the file use the `SDLFileManager`.

OBJC | SWIFT

For more information about uploading files, see the Uploading Files guide.

Using the Audio File

Now that the file is uploaded to the remote system, it can be used in various RPCs, such as `Speak`, `Alert`, and `AlertManeuver`. To use the audio file in an alert, you simply need to construct a `SDLTTSChunk` referring to the file's name.

OBJC | SWIFT

Setting Up Voice Commands

Voice commands are global commands available anywhere on the head unit to users of your app. Once the user has opened your SDL app (i.e. your SDL app has left the HMI state of `NONE`) they have access to the voice commands you have setup. Your app will be notified when a voice command has been triggered even if the SDL app has been backgrounded.

> **NOTE**
>
> The head unit manufacturer will determine how these voice commands are triggered, and some head units will not support voice commands.

> **NOTE**
>
> On Manticore, voice commands are viewed and activated by a tab in the right hand section, not through a microphone.

You have the ability to create voice command shortcuts to your Main Menu cells which we highly recommended that you implement. Global voice commands should be created for functions that you wish to make available as voice commands that are **not** available as menu cells. We recommend creating global voice commands for common actions such as the actions performed by your Soft Buttons.

Creating Voice Commands

To create voice commands, you simply create and set `SDLVoiceCommand` objects to the `voiceCommands` array on the screen manager.

OBJC | SWIFT

Unsupported Voice Commands

The library automatically filters out empty strings and whitespace-only strings from a voice command's array of strings. For example, if a voice command has the following array values: `[" ", "CommandA", "", "Command A"]` the library will filter it to: `["CommandA", "Command A"]`.

If you provide an array of voice commands which only contains empty string and whitespace-only strings across all of the voice commands, the upload request will be aborted and the previous voice commands will remain available.

Duplicate Strings in Voice Commands

DUPLICATES BETWEEN DIFFERENT COMMANDS

Voice commands that are sent with duplicate strings in different voice commands, such as:

```
{
      Command1: ["Command A", "Command B"],
      Command2: ["Command B", "Command C"],
      Command3: ["Command D", "Command E"]
}
```

Then the manager will abort the upload request. The previous voice commands will remain available.

DUPLICATES IN THE SAME COMMAND

If any individual voice command contains duplicate strings, they will be reduced to one. For example, if the voice commands to be sent are:

```
{
      Command1: ["Command A", "Command A", "Command B"],
      Command2: ["Command C", "Command D"]
}
```

Then the manager will strip the duplicates to:

```
{
      Command1: ["Command A", "Command B"],
      Command2: ["Command C", "Command D"]
}
```

Deleting Voice Commands

To delete previously set voice commands, you just have to set an empty array to the `voiceCommands` array on the screen manager.

OBJC | SWIFT

> **NOTE**
>
> Setting voice command strings composed only of whitespace characters will be considered invalid (e.g. `" "`) and your request will be aborted by the module.

Using RPCs

If you wish to do this without the aid of the screen manager, you can create `SDLAddCommand` objects without the `menuParams` parameter to create global voice commands.

Getting Microphone Audio

Capturing in-car audio allows developers to interact with users by requesting raw audio data provided to them from the car's microphones. In order to gather the raw audio from the vehicle, you must leverage the `SDLPerformAudioPassThru` RPC.

SDL does not support automatic speech cancellation detection, so if this feature is desired, it is up to the developer to implement. The user may press an "OK" or "Cancel" button, the dialog may timeout, or you may close the dialog with `SDLEndAudioPassThru`.

> **NOTE**
>
> SDL does not support an open microphone. However, SDL is working on wake-word support in the future. You may implement a voice command and start an audio pass thru session when that voice command occurs.

NOTE

Manticore does not currently support the `PerformAudioPassThru` RPC used for getting microphone audio.

Starting Audio Capture

Before you start an audio capture session you need to find out what audio pass thru capabilities the module supports. You can then use that information to start an audio pass thru session.

Getting the Supported Capabilities

You must use a sampling rate, bit rate, and audio type supported by the module. Once you have successfully connected to the module, you can access these properties on the `SDLManager.systemCapabilityManager` instance.

OBJC | SWIFT

The module may return one or multiple supported audio pass thru capabilities. Each capability will have the following properties:

AUDIO PASS THRU CAPABILITY	PARAMETER NAME	DESCRIPTION
Sampling Rate	samplingRate	The sampling rate
Bits Per Sample	bitsPerSample	The sample depth in bits
Audio Type	audioType	The audio type

Sending the Audio Capture Request

To initiate audio capture, first construct a `SDLPerformAudioPassThru` request.

OBJC | SWIFT

Gathering Audio Data

SDL provides audio data as fast as it can gather it and sends it to the developer in chunks. In order to retrieve this audio data, the developer must add a handler to the `SDLPerformAudioPassThru`.

> **NOTE**
>
> This audio data is only the current chunk of audio data, so the app is in charge of saving previously retrieved audio data.

OBJC | SWIFT

FORMAT OF AUDIO DATA

The format of audio data is described as follows:

- It does not include a header (such as a RIFF header) at the beginning.
- The audio sample is in linear PCM format.
- The audio data includes only one channel (i.e. monaural).
- For bit rates of 8 bits, the audio samples are unsigned. For bit rates of 16 bits, the audio samples are signed and are in little-endian.

Ending Audio Capture

`SDLPerformAudioPassThru` is a request that works in a different way than other RPCs. For most RPCs, a request is followed by an immediate response, with whether that RPC was successful or not. This RPC, however, will only send out the response when the audio pass thru has ended.

Audio capture can be ended four ways:

1. The audio pass thru has timed out.
    - If the audio pass thru surpasses the timeout duration, this request will be ended with a `resultCode` of `SUCCESS`. You should handle the audio pass thru as though it was successful.
2. The audio pass thru was closed due to user pressing "Cancel" (or other head-unit provided cancellation button).
    - If the audio pass thru was displayed, and the user pressed the "Cancel" button, you will receive a `resultCode` of `ABORTED`. You should ignore the audio pass thru.
3. The audio pass thru was closed due to user pressing "Done" (or other head-unit provided completion button).

- If the audio pass thru was displayed and the user pressed the "Done" button, you will receive a `resultCode` of `SUCCESS`. You should handle the audio pass thru as though it was successful.

4. The audio pass thru was ended due to a request from the app for it to end.
    - If the audio pass thru was displayed, but you have established on your own that you no longer need to capture audio data, you can send an `SDLEndAudioPassThru` RPC. You will receive a `resultCode` of `SUCCESS`. Depending on the reason that you sent the `SDLEndAudioPassThru` RPC, you can choose whether or not to handle the audio pass thru as though it were successful. See Manually Stopping Audio Capture below for more details.

Manually Stopping Audio Capture

To force stop audio capture, simply send an `SDLEndAudioPassThru` request. Your `SDLPerformAudioPassThru` request will receive response with a `resultCode` of `SUCCESS` when the audio pass thru has ended.

OBJC | SWIFT

Handling the Response

To process the response received from an ended audio capture, make sure that you are listening to the `SDLPerformAudioPassThru` response. If the response has a successful result, all of the audio data for the audio pass thru has been received and is ready for processing.

Batch Sending RPCs

There are two ways to send multiple requests to the head unit: concurrently and sequentially. Which method you should use depends on the type of RPCs being sent. Concurrently sent requests might finish in a random order and should only be used when

none of the requests in the group depend on the response of another, such as when subscribing to several hard buttons. Sequentially sent requests only send the next request in the group when a response has been received for the previously sent RPC. Requests should be sent sequentially when you need to know the result of a previous request before sending the next, like when sending the several different requests needed to create a menu.

Both methods have optional progress and completion handlers. Use the `progressHandler` to check the status of each sent RPC; it will tell you if there was an error sending the request and what percentage of the group has completed sending. The optional `completionHandler` is called when all RPCs in the group have been sent. Use it to check if all of the requests have been sent successfully or not.

Sending Concurrent Requests

When you send multiple RPCs concurrently, it will not wait for the response of the previous RPC before sending the next one. Therefore, there is no guarantee that responses will be returned in order, and you will not be able to use information sent in a previous RPC for a later RPC.

OBJC | SWIFT

Sending Sequential Requests

Requests sent sequentially are sent in a set order. The next request is only sent when a response has been received for the previously sent request.

The code example below shows how to create a perform interaction choice set. When creating a perform interaction choice set, the `SDLPerformInteraction` RPC can only be sent after the `SDLCreateInteractionChoiceSet` RPC has been registered by Core, which is why the requests must be sent sequentially.

OBJC | SWIFT

Retrieving Vehicle Data

You can use the `SDLGetVehicleData` and `SDLSubscribeVehicleData` RPC requests to get vehicle data. Each vehicle manufacturer decides which data it will expose and to whom they will expose it. Please check the response from Core to find out which data you will have permission to access. Additionally, be aware that the user may have the ability to disable vehicle data access through the settings menu of their head unit. It may be possible to access vehicle data when the `hmiLevel` is `NONE` (i.e. the user has not opened your SDL app) but you will have to request this permission from the vehicle manufacturer.

NOTE

You will only have access to vehicle data that is allowed to your `appName` and `appId` combination. Permissions will be granted by each OEM separately. See Understanding Permissions for more details.

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Acceleration Pedal Position	accPedalPosition	Accelerator pedal position (percentage depressed)		
Airbag Status	airbagStatus	Status of each of the airbags in the vehicle: yes, no, no event, not supported, fault		
Belt Status	beltStatus	The status of each of the seat belts: no, yes, not supported, fault, or no event		
Body Information	bodyInformation	Door ajar status for each door. Roof status. Trunk & hood Status. The Ignition status. The ignition stable status. The park brake active status		
Climate Data	climateData	Information about cabin temperature, atmospheric pressure, and external temperature	RPC v7.1+	

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Cloud App Vehicle Id	cloudAppVehicleID	The id for the vehicle when connecting to cloud applications	RPC v5.1+	
Cluster Mode Status	clusterModeStatus	Whether or not the power mode is active. The power mode qualification status: power mode undefined, power mode evaluation in progress, not defined, power mode ok. The car mode status: normal, factory, transport, or crash. The power mode status: key out, key recently out, key approved, post accessory, accessory, post ignition, ignition on, running, crank		

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Device Status	deviceStatus	Contains information about the smartphone device. Is voice recognition on or off, has a bluetooth connection been established, is a call active, is the phone in roaming mode, is a text message available, the battery level, the status of the mono and stereo output channels, the signal level, the primary audio source, whether or not an emergency call is currently taking place		
Driver Braking	driverBraking	The status of the brake pedal: yes, no, no event, fault, not supported		

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
E-Call Information	eCallInfo	Information about the status of an emergency call		
Electronic Parking Brake Status	electronicParkingBrakeStatus	The status of the electronic parking brake. Available states: closed, transition, open, drive active, fault	RPC v5.0+	
Emergency event	emergencyEvent	The type of emergency: frontal, side, rear, rollover, no event, not supported, fault. Fuel cutoff status: normal operation, fuel is cut off, fault. The roll over status: yes, no, no event, not supported, fault. The maximum change in velocity. Whether or not multiple emergency events have occurred		

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Engine Oil Life	engineOilLife	The estimated percentage (0% - 100%) of remaining oil life of the engine	RPC v5.0+	
Engine Torque	engineTorque	Torque value for engine (in Nm) on non-diesel variants		
External Temperature	externalTemperature	The external temperature in degrees celsius		RPC v7.1
Fuel Level	fuelLevel	The fuel level in the tank (percentage)		RPC v7.0
Fuel Level State	fuelLevel_State	The fuel level state: Unknown, Normal, Low, Fault, Alert, or Not Supported		RPC v7.0
Fuel Range	fuelRange	The estimate range in KM the vehicle can travel based on fuel level and consumption. As of RPC 7.0, this also contains Fuel Level and Fuel Level State information.	RPC v5.0+	

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Gear Status	gearStatus	Includes information about the transmission, the user's selected gear, and the actual gear of the vehicle.	RPC v7.0+	
GPS	gps	Longitude and latitude, current time in UTC, degree of precision, altitude, heading, speed, satellite data vs dead reckoning, and supported dimensions of the GPS		
Hands Off Steering	handsOffSteering	Status of hands on steering wheels capability	RPC v7.0+	

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Head Lamp Status	headLampStatus	Status of the head lamps: whether or not the low and high beams are on or off. The ambient light sensor status: night, twilight 1, twilight 2, twilight 3, twilight 4, day, unknown, invalid		
Instant Fuel Consumption	instantFuelConsumption	The instantaneous fuel consumption in microlitres		
My Key	myKey	Information about whether or not the emergency 911 override has been activated		
Odometer	odometer	Odometer reading in km		

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
PRNDL	prndl	The selected gear the car is in: park, reverse, neutral, drive, sport, low gear, first, second, third, fourth, fifth, sixth, seventh or eighth gear, unknown, or fault		RPC v7.0
RPM	rpm	The number of revolutions per minute of the engine		
Seat Occupancy	seatOccupancy	The status of the seats that show whether each seat is occupied and belted or not	RPC v7.1+	
Speed	speed	Speed in KPH		
Stability Control Status	stabilityControlsStatus	Status of the vehicle's stability control and trailer sway control	RPC v7.0+	
Steering Wheel Angle	steeringWheelAngle	Current angle of the steering wheel (in degrees)		

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Tire Pressure	tirePressure	Tire status of each wheel in the vehicle: normal, low, fault, alert, or not supported. Warning light status for the tire pressure: off, on, flash, or not used		
Turn Signal	turnSignal	The status of the turn signal. Available states: off, left, right, both	RPC v5.0+	
VIN	vin	The Vehicle Identification Number		
Window Status	windowStatus	An array of window locations and approximate position	RPC v7.0+	

VEHICLE DATA	PARAMETER NAME	DESCRIPTION	RPC VERSION	DEPRECATED
Wiper Status	wiperStatus	The status of the wipers: off, automatic off, off moving, manual interaction off, manual interaction on, manual low, manual high, manual flick, wash, automatic low, automatic high, courtesy wipe, automatic adjust, stalled, no data exists		

One-Time Vehicle Data Retrieval

To get vehicle data a single time, use the `SDLGetVehicleData` RPC.

OBJC | SWIFT

Subscribing to Vehicle Data

Subscribing to vehicle data allows you to get notifications whenever new data is available. You should not rely upon getting this data in a consistent manner. New vehicle data is available roughly every second but notification timing can vary between modules.

> **NOTE**
>
> Please note that if you are integrating an sdl_ios version less than v6.3, the following example code will not work. We recommend updating to the latest release version.

First, register to observe the `SDLDidReceiveVehicleDataNotification` notification:

OBJC | SWIFT

Second, send the `SubscribeVehicleData` request:

OBJC | SWIFT

Third, react to the notification when new vehicle data is received:

OBJC | SWIFT

Unsubscribing from Vehicle Data

We suggest that you only subscribe to vehicle data as needed. To stop listening to specific vehicle data use the `SDLUnsubscribeVehicleData` RPC.

OBJC | SWIFT

OEM-Specific Vehicle Data

OEM applications can access additional vehicle data published by their systems that is not available via the SDL vehicle data APIs. This data is accessed using the same SDL vehicle data RPCs, but instead of requesting a certain type of SDL-specified data, you must

request data using a custom vehicle data name. The type of object returned is up to the OEM and must be parsed manually.

> **NOTE**
>
> This feature is only for OEM-created applications and is not permitted for 3rd-party use.

Requesting One-Time OEM-Specific Vehicle Data

Below is an example of requesting a custom piece of vehicle data with the name `OEM-X-Vehicle-Data`. To adapt this for subscriptions instead, you must look at the section **Subscribing to Vehicle Data** above and adapt the example for subscribing to custom vehicle data based on what you see in the examples below.

OBJC | SWIFT

Remote Control Vehicle Features

The remote control framework allows apps to control modules such as climate, radio, seat, lights, etc., within a vehicle. Newer head units can support multi-zone modules that allow customizations based on seat location.

NOTE

If you are using this feature in your app, you will most likely need to request permission from the vehicle manufacturer. Not all head units support the remote control framework and only the newest head units will support multi-zone modules.

Why Use Remote Control?

Consider the following scenarios:

- A radio application wants to use the in-vehicle radio tuner. It needs the functionality to select the radio band (AM/FM/XM/HD/DAB), tune the radio frequency or change the radio station, as well as obtain general radio information for decision making.
- A climate control application needs to turn on the AC, control the air circulation mode, change the fan speed and set the desired cabin temperature.
- A user profile application wants to remember users' favorite settings and apply it later automatically when the users get into the same/another vehicle.

Supported Modules

Currently, the remote control feature supports these modules:

REMOTE CONTROL MODULES	RPC VERSION
Climate	v4.5+
Radio	v4.5+
Seat	v5.0+
Audio	v5.0+
Light	v5.0+
HMI Settings	v5.0+

The following table lists which items are in each control module.

CLIMATE

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Climate Enable	climateEnable	on, off	Get/Set/Notification	Enabled to turn on the climate system, Disabled to turn off the climate system. All other climate items need climate enabled to work.	Since v6.0
Current Cabin Temperature	currentTemperature	N/A	Get/Notification	Read only, value range depends on OEM	Since v4.5
Desired Cabin Temperature	desiredTemperature	N/A	Get/Set/Notification	Value range depends on OEM	Since v4.5
AC Setting	acEnable	on, off	Get/Set/Notification		Since v4.5
AC MAX Setting	acMaxEnable	on, off	Get/Set/Notification		Since v4.5
Air Recirculation Setting	circulateAirEnable	on, off	Get/Set/Notification		Since v4.5

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Auto AC Mode Setting	autoModeEnable	on, off	Get/Set/Notification		Since v4.5
Defrost Zone Setting	defrostZone	front, rear, all, none	Get/Set/Notification		Since v4.5
Dual Mode Setting	dualModeEnable	on, off	Get/Set/Notification		Since v4.5
Fan Speed Setting	fanSpeed	0%-100%	Get/Set/Notification		Since v4.5
Ventilation Mode Setting	ventilationMode	upper, lower, both, none	Get/Set/Notification		Since v4.5
Heated Steering Wheel Enabled	heatedSteeringWheelEnable	on, off	Get/Set/Notification		Since v5.0
Heated Windshield Enabled	heatedWindshieldEnable	on, off	Get/Set/Notification		Since v5.0
Heated Rear Window Enabled	heatedRearWindowEnable	on, off	Get/Set/Notification		Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Heated Mirrors Enabled	heatedMirrorsEnable	on, off	Get/Set/Notification		Since v5.0

RADIO

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Radio Enabled	radioEnable	true, false	Get/Set/Notification	Read only, all other radio control items need radio enabled to work	Since v4.5
Radio Band	band	AM, FM, XM	Get/Set/Notification		Since v4.5
Radio Frequency	frequencyInteger / frequencyFraction	0-1710, 0-9	Get/Set/Notification	Value range depends on band	Since v4.5
Radio RDS Data	rdsData	RdsData struct	Get/Notification	Read only	Since v4.5
Available HD Channels	availableHdChannels	Array size 0-8, values 0-7	Get/Notification	Read only	Since v6.0, replaces availableHDs
Available HD Channels (DEPRECATED)	availableHDs	1-7 (Deprecated in v6.0) (1-3 before v5.0)	Get/Notification	Read only	Since v4.5, updated in v5.0, deprecated in v6.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Current HD Channel	hdChannel	0-7 (1-3 before v.5.0) (1-7 between v.5.0-6.0)	Get/Set/Notification		Since v4.5, updated in v5.0, updated in v6.0
Radio Signal Strength	signalStrength	0-100%	Get/Notification	Read only	Since v4.5
Signal Change Threshold	signalStrengthThreshold	0-100%	Get/Notification	Read only	Since v4.5
Radio State	state	Acquiring, acquired, multicast, not_found	Get/Notification	Read only	Since v4.5
SIS Data	sisData	SisData struct	Get/Notification	Read only	Since v5.0

SEAT

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Seat Heating Enabled	heatingEnabled	true, false	Get/Set/Notification	Indicates whether heating is enabled for a seat	Since v5.0
Seat Cooling Enabled	coolingEnabled	true, false	Get/Set/Notification	Indicates whether cooling is enabled for a seat	Since v5.0
Seat Heating level	heatingLevel	0-100%	Get/Set/Notification	Level of the seat heating	Since v5.0
Seat Cooling level	coolingLevel	0-100%	Get/Set/Notification	Level of the seat cooling	Since v5.0
Seat Horizontal Position	horizontalPosition	0-100%	Get/Set/Notification	Adjust a seat forward/backward, 0 means the nearest position to the steering wheel, 100% means the furthest position from the steering wheel	Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Seat Vertical Position	verticalPosition	0-100%	Get/Set/Notification	Adjust seat height (up or down) in case there is only one actuator for seat height, 0 means the lowest position, 100% means the highest position	Since v5.0
Seat-Front Vertical Position	frontVerticalPosition	0-100%	Get/Set/Notification	Adjust seat front height (in case there are two actuators for seat height), 0 means the lowest position, 100% means the highest position	Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Seat-Back Vertical Position	backVertical Position	0-100%	Get/Set/Notification	Adjust seat back height (in case there are two actuators for seat height), 0 means the lowest position, 100% means the highest position	Since v5.0
Seat Back Tilt Angle	backTiltAngle	0-100%	Get/Set/Notification	Backrest recline, 0 means the angle that back top is nearest to the steering wheel, 100% means the angle that back top is furthest from the steering wheel	Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Head Support Horizontal Position	headSupportHorizontalPosition	0-100%	Get/Set/Notification	Adjust head support forward/backward, 0 means the nearest position to the front, 100% means the furthest position from the front	Since v5.0
Head Support Vertical Position	headSupportVerticalPosition	0-100%	Get/Set/Notification	Adjust head support height (up or down), 0 means the lowest position, 100% means the highest position	Since v5.0
Seat Massaging Enabled	massageEnabled	true, false	Get/Set/Notification	Indicates whether massage is enabled for a seat	Since v5.0
Massage Mode	massageMode	MassageModeData struct	Get/Set/Notification	List of massage mode of each zone	Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Massage Cushion Firmness	massageCushionFirmness	MassageCushionFirmness struct	Get/Set/Notification	List of firmness of each massage cushion	Since v5.0
Seat memory	memory	SeatMemoryAction struct	Get/Set/Notification	Seat memory	Since v5.0

AUDIO

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Audio Volume	volume	0%-100%	Get/Set/Notification	The audio source volume level	Since SDL v5.0
Audio Source	source	PrimaryAudioSource enum	Get/Set/Notification	Defines one of the available audio sources	Since SDL v5.0
Keep Context	keepContext	true, false	Set only	Controls whether the HMI will keep the current application context or switch to the default media UI/APP associated with the audio source	Since SDL v5.0
Equalizer Settings	equalizerSettings	EqualizerSettings struct	Get/Set/Notification	Defines the list of supported channels (band) and their current/desired settings on HMI	Since SDL v5.0

LIGHT

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Light State	lightState	Array of LightState struct	Get/Set/Notification		Since SDL v5.0

HMI SETTINGS

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Display Mode	displayMode	Day, Night, Auto	Get/Set/Notification	Current display mode of the HMI display	Since SDL v5.0
Distance Unit	distanceUnit	Miles, Kilometers	Get/Set/Notification	Distance Unit used in the HMI (for maps/tracking distances)	Since SDL v5.0
Temperature Unit	temperatureUnit	Fahrenheit, Celsius	Get/Set/Notification	Temperature Unit used in the HMI (for temperature measuring systems)	Since SDL v5.0

Remote Control Button Presses

The remote control framework also allows mobile applications to send simulated button press events for the following common buttons in the vehicle.

RC MODULE	CONTROL BUTTON
Climate	AC
	AC MAX
	RECIRCULATE
	FAN UP
	FAN DOWN
	TEMPERATURE UP
	TEMPERATURE DOWN
	DEFROST
	DEFROST REAR
	DEFROST MAX
	UPPER VENT
	LOWER VENT
Radio	VOLUME UP
	VOLUME DOWN
	EJECT
	SOURCE

RC MODULE	CONTROL BUTTON
	SHUFFLE
	REPEAT

Integration

For remote control to work, the head unit must support SDL RPC v4.4+. In addition, your app's `appType` / `additionalAppTypes` must include `REMOTE_CONTROL`.

Multiple Modules (RPC v6.0+)

Each module type can have multiple modules in RPC v6.0+. In previous versions, only one module was available for each module type. A specific module is controlled using the unique id assigned to the module. When sending remote control RPCs to a RPC v6.0+ head unit, the `moduleInfo.moduleId` must be stored and provided to control the desired module. If no `moduleId` is set, the HMI will use the default module of that module type. When connected to <6.0 systems, the `moduleInfo` struct will be `nil`, and only the default module will be available for control.

Getting Remote Control Module Information

Prior to using any remote control RPCs, you must check that the head unit has the remote control capability. As you will encounter head units that do *not* support remote control, or head units that do not give your application permission to read and write remote control data, this check is important.

> **NOTE**
>
> This check can be performed once your SDL app has left the HMI state of `NONE`. More information on how to monitor the HMI status can be found in the Understanding Permissions guide.

When connected to head units supporting RPC v6.0+, you should save this information for future use. The `moduleId` contained within the `moduleInfo` struct on each capability is necessary to control that module.

OBJC | SWIFT

GETTING MODULE DATA LOCATION AND SERVICE AREAS (RPC V6.0+)

With the saved remote control capabilities struct you can get the location of the each module and the area that it services. This will map to the `grid` graphic below. This information is useful for creating a custom UI.

> **NOTE**
>
> This data is only available when connected to SDL RPC v6.0+ systems. On previous systems, only one module per module type was available, so the module's location didn't matter. You will not be able to build a custom UI for those cases and should use a generic UI instead.

OBJC | SWIFT

You can also get an array of seats in the `SDLSeatLocationCapability.seats` array. Each `SDLSeatLocation` object within the `seats` array will have a `grid` parameter. The `grid` will tell you the location of that particular seat in the vehicle (See the graphic below).

OBJC | SWIFT

The Grid

The `grid` system starts with the front left corner of the bottom level of the vehicle being `(col=0, row=0, level=0)`. For example, assuming a vehicle manufactured for sale in the United States with three seats in the backseat, `(0, 0, 0)` would be the drivers' seat. The front passenger location would be at `(2, 0, 0)` and the rear middle seat would be at `(1, 1, 0)`. The `colspan` and `rowspan` properties tell you how many rows and columns that module or seat takes up. The `level` property tells you how many decks the vehicle has (i.e. a double-decker bus would have 2 levels).

Col+
Row+
(0,0,0)
(2,0,0)
(0,1,0)
(2,1,0)

	COL=0	COL=1	COL=2
row=0	driver's seat: {col=0, row=0, level=0, colspan=1, rowspan=1, levelspan=1}		front passenger's seat : {col=2, row=0, level=0, colspan=1, rowspan=1, levelspan=1}
row=1	rear-left seat : {col=0, row=1, level=0, colspan=1, rowspan=1, levelspan=1}	rear-middle seat : {col=1, row=1, level=0, colspan=1, rowspan=1, levelspan=1}	rear-right seat : {col=2, row=1, level=0, colspan=1, rowspan=1, levelspan=1}

Getting Module Data

Seat location does not affect the ability to get data from a module. Once you know you have permission to use the remote control feature and you have `moduleId`s (when connected to RPC v6.0+ systems), you can retrieve the data for any module. The following code is an example of how to subscribe to the data of a climate module.

When connected to head units that only support RPC versions older than v6.0, there can only be one module for each module type (e.g. there can only be one climate module, light module, radio module, etc.), so you will not need to pass a `moduleId`.

SUBSCRIBING TO MODULE DATA

You can either subscribe to module data or receive it one time. If you choose to subscribe to module data you will receive continuous updates on the vehicle data you have subscribed to.

> **NOTE**
>
> Subscribing to the `OnInteriorVehicleData` notification must be done before sending the `SDLGetInteriorVehicleData` request.

OBJC | SWIFT

After you subscribe to the `SDLDidReceiveInteriorVehicleDataNotification` you must also subscribe to the module you wish to receive updates for. Subscribing to a module will send a notification when that particular module is changed.

RPC < v6.0

OBJC | SWIFT

RPC v6.0+

OBJC | SWIFT

After you subscribe to the `InteriorVehicleDataNotification` you must also subscribe to the module you wish to receive updates for. Subscribing to a module will send a notification when that particular module is changed.

GETTING ONE-TIME DATA

To get data from a module without subscribing send a `SDLGetInteriorVehicleData` request with the `subscribe` flag set to `false`.

RPC < v6.0

OBJC | SWIFT

RPC v6.0+

OBJC | SWIFT

Setting Module Data

Not only do you have the ability to get data from these modules, but, if you have the right permissions, you can also set module data.

SETTING THE USER'S SEAT (RPC V6.0+)

Before you attempt to take control of any module, you should have your user select their seat location as this affects which modules they have permission to control. You may wish to show the user a map or list of all available seats in your app in order to ask them where they are located. See Getting Module Data Location and Service Areas for information useful in creating a custom UI showing module location and service area. The following example is only meant to show you how to access the available data and not how to build your UI/UX.

When the user selects their seat, you must send an `SDLSetGlobalProperties` RPC with the appropriate `userLocation` property in order to update that user's location within the vehicle (The default seat location is `Driver`).

OBJC | SWIFT

GETTING CONSENT TO CONTROL A MODULE (RPC V6.0+)

Some OEMs may wish to ask the driver for consent before a user can control a module. The `SDLGetInteriorVehicleDataConsent` RPC will alert the driver in some OEM head units if the module is not free (another user has control) and `allowMultipleAccess` (multiple users can access/set the data at the same time) is `true`. The `allowMultipleAccess` property is part of the `moduleInfo` in the module object.

Check the `allowed` property in the `SDLGetInteriorVehicleDataConsentResponse` to see what modules can be controlled. Note that the order of the `allowed` array is 1-1 with the `moduleIds` array you passed into the `SDLGetInteriorVehicleDataConsent` RPC.

> **NOTE**
>
> You should always try to get consent before setting any module data. If consent is not granted you should not attempt to set any module's data.

OBJC | SWIFT

CONTROLLING A MODULE

Below is an example of setting climate control data. It is likely that you will not need to set all the data as in the code example below. When connected to RPC v6.0+ systems, you must set the `moduleId` in `SDLSetInteriorVehicleData.moduleData`. When connected to < v6.0 systems, there is only one module per module type, so you must only pass the type of the module you wish to control.

When you received module information above in **Getting Remote Control Module Information** on RPC v6.0+ systems, you received information on the `location` and `serviceArea` of the module. The permission area of a module depends on that `serviceArea`. The `location` of a module is like the `seats` array: it maps to the `grid` to tell you the physical location of a particular module. The `serviceArea` maps to the grid to show how far that module's scope reaches.

For example, a radio module usually serves all passengers in the vehicle, so its service area will likely cover the entirety of the vehicle grid, while a climate module may only cover a passenger area and not the driver or the back row. If a `serviceArea` is not included, it is assumed that the `serviceArea` is the same as the module's `location`. If neither is included, it is assumed that the `serviceArea` covers the whole area of the vehicle. If a user is not sitting within the `serviceArea`'s `grid`, they will not receive permission to control that module (attempting to set data will fail).

RPC < v6.0

OBJC | SWIFT

RPC v6.0+

OBJC | SWIFT

BUTTON PRESSES

Another unique feature of remote control is the ability to send simulated button presses to the associated modules, imitating a button press on the hardware itself. Simply specify the module, the button, and the type of press you would like to simulate.

RPC < v6.0

OBJC | SWIFT

RPC v6.0+

OBJC | SWIFT

```
SDLButtonPress *buttonPress = [[SDLButtonPress alloc]
initWithButtonName:SDLButtonNameTempUp moduleType:SDLModuleTypeClimate
moduleId:@"<#ModuleID#>" buttonPressMode:SDLButtonPressModeShort];

[self.sdlManager sendRequest:buttonPress withResponseHandler:^(__kindof
SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse * _Nullable
response, NSError * _Nullable error) {
    if(!response.success) { return; }
}];
```

RELEASING THE MODULE (RPC V6.0+)

When the user no longer needs control over a module, you should release the module so other users can control it. If you do not release the module, other users who would otherwise be able to control the module may be rejected from doing so.

OBJC | SWIFT

```
SDLReleaseInteriorVehicleDataModule *releaseInteriorVehicleDataModule =
[[SDLReleaseInteriorVehicleDataModule alloc] initWithModuleType:<#ModuleType#>
moduleId:@"<#ModuleID#>"];
[self.sdlManager sendRequest:releaseInteriorVehicleDataModule
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request, __kindof
SDLRPCResponse * _Nullable response, NSError * _Nullable error) {
```

```
    if(!response.success) { return; }
    <#Module Was Released#>
}];
```

Creating an App Service (RPC v5.1+)

App services is a powerful feature enabling both a new kind of vehicle-to-app communication and app-to-app communication via SDL.

App services are used to publish navigation, weather and media data (such as temperature, navigation waypoints, or the current playlist name). This data can then be used by both the vehicle head unit and, if the publisher of the app service desires, other SDL apps.

Vehicle head units may use these services in various ways. One app service for each type will be the "active" service to the module. For media, for example, this will be the media app that the user is currently using or listening to. For navigation, it would be a navigation app that the user is using to navigate. For weather, it may be the last used weather app, or a user-selected default. The system may then use that service's data to perform various actions (such as navigating to an address with the active service or to display the temperature as provided from the active weather service).

An SDL app can also subscribe to a published app service. Once subscribed, the app will be sent the new data when the app service publisher updates its data. To find out more about how to subscribe to an app service check out the Using App Services guide. Subscribed apps can also send certain RPCs and generic URI-based actions (see the section Supporting Service RPCs and Actions below) to your service.

Currently, there is no high-level API support for publishing an app service, so you will have to use raw RPCs for all app service related APIs.

Using an app service is covered in another guide.

App Service Types

Apps are able to declare that they provide an app service by publishing an app service manifest. Three types of app services are currently available and more will be made available over time. The currently available types are: Media, Navigation, and Weather. An app may publish multiple services (one for each of the different service types) if desired.

Publishing an App Service

Publishing a service is a multi-step process. First, you need to create your app service manifest. Second, you will publish your app service to the module. Third, you will publish the service data using `OnAppServiceData`. Fourth, you must listen for data requests and respond accordingly. Fifth, if your app service supports handling of RPCs related to your service you must listen for these RPC requests and handle them accordingly. Sixth, optionally, you can support URI-based app actions. Finally, if necessary, you can you update or delete your app service manifest.

> **NOTE**
>
> Please note that if you are integrating an sdl_ios version less than v6.3, the example code in this guide will not work. We recommend updating to the latest release version.

1. Creating an App Service Manifest

The first step to publishing an app service is to create an `SDLAppServiceManifest` object. There is a set of generic parameters you will need to fill out as well as service type specific parameters based on the app service type you are creating.

OBJC | SWIFT

CREATING A MEDIA SERVICE MANIFEST

Currently, there's no information you have to provide in your media service manifest! You'll just have to create an empty media service manifest and set it into your general app service manifest.

OBJC | SWIFT

CREATING A NAVIGATION SERVICE MANIFEST

You will need to create a navigation manifest if you want to publish a navigation service. You will declare whether or not your navigation app will accept waypoints. That is, if your app will support receiving *multiple* points of navigation (e.g. go to this McDonalds, then this Walmart, then home).

OBJC | SWIFT

CREATING A WEATHER SERVICE MANIFEST

You will need to create a weather service manifest if you want to publish a weather service. You will declare the types of data your service provides in its `SDLWeatherServiceData`.

OBJC | SWIFT

2. Publish Your Service

Once you have created your service manifest, publishing your app service is simple.

OBJC | SWIFT

Once you have your publish app service response, you will need to store the information provided in its `appServiceRecord` property. You will need the information later when you want to update your service's data.

WATCHING FOR APP RECORD UPDATES

As noted in the introduction to this guide, one service for each type may become the "active" service. If your service is the active service, your `SDLAppServiceRecord` parameter `serviceActive` will be updated to note that you are now the active service.

After the initial app record is passed to you in the `SDLPublishAppServiceResponse`, you will need to be notified of changes in order to observe whether or not you have become the active service. To do so, you will have to observe the new `SDLSystemCapabilityTypeAppServices` using `GetSystemCapability` and `OnSystemCapability`.

For more information, see the Using App Services guide and go to the **Getting and Subscribing to Services** section.

3. Update Your Service's Data

After your service is published, it's time to update your service data. First, you must send an `onAppServiceData` RPC notification with your updated service data. RPC notifications are different than RPC requests in that they will not receive a response from the connected head unit , and must use a different `SDLManager` method call to send.

> **NOTE**
>
> You should only update your service's data when you are the active service; service consumers will only be able to see your data when you are the active service.

First, you will have to create an `SDLMediaServiceData`, `SDLNavigationServiceData` or `SDLWeatherServiceData` object with your service's data. Then, add that service-specific data object to an `SDLAppServiceData` object. Finally, create an `SDLOnAppServiceData` notification, append your `SDLAppServiceData` object, and send it.

MEDIA SERVICE DATA

OBJC | SWIFT

NAVIGATION SERVICE DATA

OBJC | SWIFT

WEATHER SERVICE DATA

OBJC | SWIFT

4. Handling App Service Subscribers

If you choose to make your app service available to other apps, you will have to handle requests to get your app service data when a consumer requests it directly.

Handling app service subscribers is a two step process. First, you must register for notifications from the subscriber. Then, when you get a request, you will either have to send a response to the subscriber with the app service data or if you have no data to send, send a response with a relevant failure result code.

LISTENING FOR REQUESTS

First, you will need to subscribe to `GetAppServiceDataRequest` notifications.. Then, when you get the request, you will need to respond with your app service data. Therefore, you will need to store your current service data after the most recent update using `OnAppServiceData` (see the section Update Your Service's Data).

OBJC SWIFT

```
__weak typeof(self) weakSelf = self;
[self.sdlManager subscribeToRPC:SDLDidReceiveGetAppServiceDataRequest
withBlock:^(__kindof SDLRPCMessage * _Nonnull message) {
    SDLGetAppServiceData *getAppServiceRequest = message;

    // Send a response
    SDLGetAppServiceDataResponse *response = [[SDLGetAppServiceDataResponse
alloc] initWithAppServiceData:<#Your App Service Data#>];
    response.correlationID = getAppServiceRequest.correlationID;
    response.success = @YES;
    response.resultCode = SDLResultSuccess;
    response.info = @"<#Use to provide more information about an error#>";
    [weakSelf.sdlManager sendRPC:response];
}];
```

Supporting Service RPCs and Actions

5. Service RPCs

Certain RPCs are related to certain services. The chart below shows the current relationships:

MEDIA	NAVIGATION	WEATHER
ButtonPress (OK)	SendLocation	
ButtonPress (SEEKLEFT)	GetWayPoints	
ButtonPress (SEEKRIGHT)	SubscribeWayPoints	
ButtonPress (TUNEUP)	OnWayPointChange	
ButtonPress (TUNEDOWN)		
ButtonPress (SHUFFLE)		
ButtonPress (REPEAT)		

When you are the active service for your service's type (e.g. media), and you have declared that you support these RPCs in your manifest (see the section Creating an App Service Manifest), then these RPCs will be automatically routed to your app. You will have to set up notifications to be aware that they have arrived, and you will then need to respond to those requests.

OBJC SWIFT

```
SDLAppServiceManifest *manifest = [[SDLAppServiceManifest alloc] init];
// Everything else for your manifest
NSNumber *buttonPressRPCID = [[SDLFunctionID sharedInstance]
functionIdForName:SDLRPCFunctionNameButtonPress];
manifest.handledRPCs = @[buttonPressRPCID];

[self.sdlManager subscribeToRPC:SDLDidReceiveButtonPressRequest
withObserver:self selector:@selector(buttonPressRequestReceived:)];

- (void)buttonPressRequestReceived:(SDLRPCRequestNotification *)request {
    SDLButtonPress *buttonPressRequest = (SDLButtonPress *)request.request;
    // Check the request for the button name and long / short press

    // Send a response
    SDLButtonPressResponse *response = [[SDLButtonPressResponse alloc] init];
```

```
    response.correlationID = buttonPressRequest.correlationID;
    response.success = @YES;
    response.resultCode = SDLVehicleDataResultCodeSuccess;
    response.info = @"<#Use to provide more information about an error#>";

    [self.sdlManager sendRPC:response];
}
```

6. Service Actions

App actions are the ability for app consumers to use the SDL services system to send URIs to app providers in order to activate actions on the provider. Service actions are *schema-less*, i.e. there is no way to define the appropriate URIs through SDL. If you already provide actions through your app and want to expose them to SDL, or if you wish to start providing them, you will have to document your available actions elsewhere (such as your website).

If you're wondering how to get started with actions and routing, this is a very common problem in iOS! Many apps support the x-callback-URL format as a common inter-app communication method. There are also many libraries available for the purpose of supporting URL routing.

In order to support actions through SDL services, you will need to observe and respond to the `PerformAppServiceInteraction` RPC request.

OBJC | SWIFT

```
// Subscribe to PerformAppServiceInteraction requests
[self.sdlManager
subscribeToRPC:SDLDidReceivePerformAppServiceInteractionRequest
withObserver:self
selector:@selector(performAppServiceInteractionRequestReceived:)];

- (void)performAppServiceInteractionRequestReceived:(SDLRPCRequestNotification
*)notification {
    SDLPerformAppServiceInteraction *interactionRequest = notification.request;

    // If you have multiple services, this will let you know which of your services is
being addressed
    NSString *serviceID = interactionRequest.serviceID;

    // The app id of the service consumer app that sent you this message
    NSString *originAppId = interactionRequest.originApp;

    // The URL sent by the consumer. This must be something you understand, e.g. a
```

```
URL scheme call. For example, if you were YouTube, it could be a URL to play a specific video. If you were a music app, it could be a URL to play a specific song, activate shuffle / repeat, etc.
    NSURLComponents *interactionURLComponents = [NSURLComponents componentsWithString:interactionRequest.serviceUri];

    // A result you want to send to the consumer app.
    NSString *result = @"Uh oh";
    SDLPerformAppServiceInteractionResponse *response = [[SDLPerformAppServiceInteractionResponse alloc] initWithServiceSpecificResult:result];

    // These are very important, your response won't work properly without them.
    response.success = @NO;
    response.resultCode = SDLResultGenericError;
    response.correlationID = interactionRequest.correlationID;

    [self.sdlManager sendRPC:response];
}
```

Updating Your Published App Service

Once you have published your app service, you may decide to update its data. For example, if you have a free and paid tier with different amounts of data, you may need to upgrade or downgrade a user between these tiers and provide new data in your app service manifest. If desired, you can also delete your app service by unpublishing the service.

7. Updating a Published App Service Manifest (RPC v6.0+)

OBJC | SWIFT

```
SDLAppServiceManifest *manifest = [[SDLAppServiceManifest alloc] initWithAppServiceType:SDLAppServiceTypeWeather];
manifest.weatherServiceManifest = <#Updated weather service manifest#>;

SDLPublishAppService *publishServiceRequest = [[SDLPublishAppService alloc] initWithAppServiceManifest:manifest];
[self.sdlManager sendRequest:publishServiceRequest];
```

8. Unpublishing a Published App Service Manifest (RPC v6.0+)

OBJC SWIFT

```
SDLUnpublishAppService *unpublishAppService = [[SDLUnpublishAppService alloc] initWithServiceID:@"<#The serviceID of the service to unpublish#>"];
[self.sdlManager sendRequest:unpublishAppService];
```

Using App Services (RPC v5.1+)

App services is a powerful feature enabling both a new kind of vehicle-to-app communication and app-to-app communication via SDL.

App services are used to publish navigation, weather and media data (such as temperature, navigation waypoints, or the current playlist name). This data can then be used by both the vehicle head unit and, if the publisher of the app service desires, other SDL apps. Creating an app service is covered in another guide.

Vehicle head units may use these services in various ways. One app service for each type will be the "active" service to the module. For media, for example, this will be the media app that the user is currently using or listening to. For navigation, it would be a navigation app that the user is using to navigate. For weather, it may be the last used weather app, or a user-selected default. The system may then use that service's data to perform various actions (such as navigating to an address with the active service or to display the temperature as provided from the active weather service).

An SDL app can also subscribe to a published app service. Once subscribed, the app will be sent the new data when the app service publisher updates its data. This guide will cover subscribing to a service. Subscribed apps can also send certain RPCs and generic URI-based actions (see the section Sending an Action to a Service Provider, below) to your service.

Currently, there is no high-level API support for using an app service, so you will have to use raw RPCs for all app service related APIs.

Getting and Subscribing to Services

Once your app has connected to the head unit, you will first want to be notified of all available services and updates to the metadata of all services on the head unit. Second, you will narrow down your app to subscribe to an individual app service and subscribe to its data. Third, you may want to interact with that service through RPCs, or fourth, through service actions.

> **NOTE**
>
> Please note that if you are integrating an sdl_ios version less than v6.3, the example code in this guide will not work. We recommend updating to the latest release version.

1. Getting and Subscribing to Available Services

To get information on all services published on the system, as well as on changes to published services, you will use the `SystemCapabilityManager`.

OBJC | SWIFT

CHECKING THE APP SERVICE CAPABILITY

Once you've retrieved the initial list of app service capabilities or an updated list of app service capabilities, you may want to inspect the data to find what you are looking for. Below is example code with comments explaining what each part of the app service capability is used for.

OBJC | SWIFT

2. Getting and Subscribing to a Service Type's Data

Once you have information about all of the services available, you may want to view or subscribe to a service type's data. To do so, you will use the `GetAppServiceData` RPC.

Note that you will currently only be able to get data for the *active* service of the service type. You can attempt to make another service the active service by using the `PerformAppServiceInteraction` RPC, discussed below in Sending an Action to a Service Provider.

OBJC | SWIFT

Interacting with a Service Provider

Once you have a service's data, you may want to interact with a service provider by sending RPCs or actions.

3. Sending RPCs to a Service Provider

Only certain RPCs are available to be passed to the service provider based on their service type. See the Creating an App Service guide **Supporting Service RPCs and Actions** section for a chart detailing which RPCs work with which service types. The RPC can only be sent to the active service of a specific service type, not to any inactive service.

Sending an RPC works exactly the same as if you were sending the RPC to the head unit system. The head unit will simply route your RPC to the appropriate app automatically.

> **NOTE**
>
> Your app may need special permissions to use the RPCs that route to app service providers.

OBJC | SWIFT

4. Sending an Action to a Service Provider

Actions are generic URI-based strings sent to any app service (active or not). You can also use actions to request to the system that they make the service the active service for that service type. Service actions are *schema-less*, i.e. there is no way to define the appropriate URIs through SDL. The service provider must document their list of available actions elsewhere (such as their website).

OBJC | SWIFT

5. Getting a File from a Service Provider

In some cases, a service may upload an image that can then be retrieved from the module. First, you will need to get the image name from the `SDLAppServiceData` (see point 2 above). Then you will use the image name to retrieve the image data.

OBJC | SWIFT

Calling a Phone Number

The `SDLDialNumber` RPC allows you make a phone call via the user's phone. In order to dial a phone number you must be sure that the device is connected via Bluetooth (even if your device is also connected using a USB cord) for this request to work. If the phone is not connected via Bluetooth, you will receive a result of `REJECTED` from the module.

Checking Your App's Permissions

`SDLDialNumber` is an RPC that is usually restricted by OEMs. As a result, a module may reject your request if your app does not have the correct permissions. Your SDL app may

also be restricted to only being allowed to making a phone call when your app is open (i.e. the `hmiLevel` is non-`NONE`) or when it is the currently active app (i.e. the `hmiLevel` is `FULL`).

OBJC | SWIFT

Checking if the Module Supports Calling a Phone Number

Since making a phone call is a newer feature, there is a possibility that some legacy modules will reject your request because the module does not support the `SDLDialNumber` request. Once you have successfully connected to the module, you can check the module's capabilities via the `SDLManager.systemCapabilityManager` as shown in the example below. Please note that you only need to check once if the module supports calling a phone number, however you must wait to perform this check until you know that the SDL app has been opened (i.e. the `hmiLevel` is non-`NONE`).

> **NOTE**
>
> If you discover that the module does not support calling a phone number or that your app does not have the right permissions, you should disable any buttons, voice commands, menu items, etc. in your app that would send the `SDLDialNumber` request.

OBJC | SWIFT

Sending a DialNumber Request

Once you know that the module supports dialing a phone number and that your SDL app has permission to send the `SDLDialNumber` request, you can create and send the request.

> **NOTE**
>
> `SDLDialNumber` strips all characters except for `0` - `9`, `*`, `#`, `,`, `;`, and `+`.

OBJC | SWIFT

Dial Number Responses

The `SDLDialNumber` request has three possible responses that you should expect:

1. `SUCCESS` - The request was successfully sent, and a phone call was initiated by the user.
2. `REJECTED` - This can mean either:
    - The user rejected the request to make the phone call.
    - The phone is not connected to the module via Bluetooth.
3. `DISALLOWED` - Your app does not have permission to use the `SDLDialNumber` request.

Setting the Navigation Destination

The `SDLSendLocation` RPC gives you the ability to send a GPS location to the active navigation app on the module.

When using the `SDLSendLocation` RPC, you will not have access to any information about how the user interacted with this location, only if the request was successfully sent. The request will be handled by the module from that point on using the active navigation system.

Checking Your App's Permissions

The `SDLSendLocation` RPC is restricted by most OEMs. As a result, a module may reject your request if your app does not have the correct permissions. Your SDL app may also be restricted to only being allowed to send a location when your app is open (i.e. the `hmiLevel` is non-`NONE`) or when it is the currently active app (i.e. the `hmiLevel` is `FULL`).

OBJC | SWIFT

Checking if the Module Supports Sending a Location

Since some modules will not support sending a location, you should check if the module supports this feature before trying to use it. Once you have successfully connected to the module, you can check the module's capabilities via the `SDLManager.systemCapabilityManager` as shown in the example below. Please note that you only need to check once if the module supports sending a location, however you must wait to perform this check until you know that the SDL app has been opened (i.e. the `hmiLevel` is non-`NONE`).

NOTE

If you discover that the module does not support sending a location or that your app does not have the right permissions, you should disable any buttons, voice commands, menu items, etc. in your app that would send the `SDLSendLocation` request.

OBJC | SWIFT

Using Send Location

To use the `SDLSendLocation` request, you must at minimum include the longitude and latitude of the location.

OBJC | SWIFT

Checking the Result of Send Location

The `SDLSendLocation` request has three possible responses that you should expect:

1. `SUCCESS` - Successfully sent.
2. `INVALID_DATA` - The request contains invalid data and was rejected.
3. `DISALLOWED` - Your app does not have permission to use the `SDLSendLocation` request.

Getting the Navigation Destination (RPC v4.1+)

The `SDLGetWayPoints` and `SDLSubscribeWayPoints` RPCs are designed to allow you to get the navigation destination(s) from the active navigation app when the user has activated in-car navigation.

Checking Your App's Permissions

Both the `SDLGetWayPoints` and `SDLSubscribeWayPoints` RPCs are restricted by most OEMs. As a result, a module may reject your request if your app does not have the correct permissions. Your SDL app may also be restricted to only being allowed to get waypoints when your app is open (i.e. the `hmiLevel` is non-`NONE`) or when it is the currently active app (i.e. the `hmiLevel` is `FULL`).

OBJC | SWIFT

Checking if the Module Supports Waypoints

Since some modules will not support getting waypoints, you should check if the module supports this feature before trying to use it. Once you have successfully connected to the module, you can check the module's capabilities via the `SDLManager.systemCapabilityManager` as shown in the example below. Please note that you only need to check once if the module supports getting waypoints, however you must wait to perform this check until you know that the SDL app has been opened (i.e. the `hmiLevel` is non-`NONE`).

> **NOTE**
>
> If you discover that the module does not support getting navigation waypoints or that your app does not have the right permissions, you should disable any buttons, voice commands, menu items, etc. in your app that would send the `SDLGetWayPoints` or `SDLSubscribeWayPoints` requests.

OBJC | SWIFT

Subscribing to Waypoints

To subscribe to the navigation waypoints, you will have to set up your callback for whenever the waypoints are updated, then send the `SDLSubscribeWayPoints` RPC.

OBJC | SWIFT

Unsubscribing from Waypoints

To unsubscribe from waypoint data, you must send the `SDLUnsubscribeWayPoints` RPC.

> **NOTE**
>
> You do not have to unsubscribe from the `sdlManager.subscribe` method, you must simply send the unsubscribe RPC and no more callbacks will be received.

OBJC | SWIFT

One-Time Waypoints Request

If you only need waypoint data once without an ongoing subscription, you can use `SDLGetWayPoints` instead of `SDLSubscribeWayPoints`.

OBJC | SWIFT

Uploading Files

In almost all cases, you will not need to handle uploading images because the screen manager API will do that for you. There are some situations, such as VR help-lists and turn-by-turn directions, that are not currently covered by the screen manager so you will have manually upload the image yourself in those cases. For more information about uploading images, see the Uploading Images guide.

Uploading an MP3 Using the File Manager

The `SDLFileManager` uploads files and keeps track of all the uploaded files names during a session. To send data with the file manager you need to create either a `SDLFile` or `SDLArtwork` object. `SDLFile` objects are created with a local `NSURL` or `NSData`; `SDLArtwork` uses a `UIImage`.

OBJC | SWIFT

Batching File Uploads

If you want to upload a group of files, you can use the `SDLFileManager`'s batch upload methods. Once all of the uploads have completed you will be notified if any of the uploads failed. If desired, you can also track the progress of each file in the group.

OBJC | SWIFT

File Persistence

`SDLFile` and its subclass `SDLArtwork` support uploading persistent files, i.e. files that are not deleted when the car turns off. Persistence should be used for files that will be used every time the user opens the app. If the file is only displayed for short time the file should not be persistent because it will take up unnecessary space on the head unit. You can check the persistence via:

OBJC | SWIFT

> **NOTE**
>
> Be aware that persistence will not work if space on the head unit is limited. The `SDLFileManager` will always handle uploading images if they are non-existent.

Overwriting Stored Files

If a file being uploaded has the same name as an already uploaded file, the new file will be ignored. To override this setting, set the `SDLFile`'s `overwrite` property to `true`.

OBJC | SWIFT

Checking the Amount of File Storage Left

To find the amount of file storage left for your app on the head unit, use the `SDLFileManager`'s `bytesAvailable` property.

OBJC | SWIFT

Checking if a File Has Already Been Uploaded

You can check out if an image has already been uploaded to the head unit via the `SDLFileManager`'s `remoteFileNames` property.

OBJC SWIFT

```
BOOL isFileOnHeadUnit = [self.sdlManager.fileManager.remoteFileNames containsObject:<#Name Uploaded As#>];
```

Deleting Stored Files

Use the file manager's delete request to delete a file associated with a file name.

OBJC SWIFT

```
[self.sdlManager.fileManager deleteRemoteFileWithName:@"<#Name Uploaded As#>" completionHandler:^(BOOL success, NSUInteger bytesAvailable, NSError *error) {
    if (success) {
        <#File was deleted successfully#>
    }
}];
```

Batch Deleting Files

OBJC SWIFT

```
[self.sdlManager.fileManager deleteRemoteFilesWithNames:@[@"<#Name Uploaded As#>", @"<#Name Uploaded As 2#>"] completionHandler:^(NSError * _Nullable error)
{
    if (error == nil) {
        <#Images were deleted successfully#>
    }
}];
```

Uploading Images

> **NOTE**
>
> If you use the `SDLScreenManager`, image uploading for template graphics, soft buttons, and menu items is handled for you behind the scenes. However, you will still need to manually upload your images if you need images in an alert, VR help lists, turn-by-turn directions, or other features not currently covered by the `SDLScreenManager`.

You should be aware of these four things when using images in your SDL app:

1. You may be connected to a head unit that does not have the ability to display images.
2. You must upload images from your mobile device to the head unit before using them in a template.
3. Persistent images are stored on a head unit between sessions. Ephemeral images are destroyed when a session ends (i.e. when the user turns off their vehicle).
4. Images can not be uploaded when the app's `hmiLevel` is `NONE`. For more information about permissions, please review Understanding Permissions.

Checking if Graphics are Supported

Before uploading images to a head unit you should first check if the head unit supports graphics. If not, you should avoid uploading unnecessary image data. To check if graphics are supported, check the `SDLManager.systemCapabilityManager.defaultMainWindowCapability` property once the `SDLManager` has started successfully.

OBJC | SWIFT

Uploading an Image Using the File Manager

The `SDLFileManager` uploads files and keeps track of all the uploaded files names during a session. To send data with the `SDLFileManager`, you need to create either a `SDLFile` or `SDLArtwork` object. `SDLFile` objects are created with a local `NSURL` or `NSData`; `SDLArtwork` a `UIImage`.

OBJC | SWIFT

Batch File Uploads, Persistence, etc.

Similar to other files, artworks can be persistent, batched, overwrite, etc. See Uploading Files for more information.

Creating an OEM Cloud App Store (RPC v5.1+)

SDL allows OEMs to offer an app store that lets users browse and install remote cloud apps. If the cloud app requires users to login with their credentials, the app store can use an authentication token to automatically login users after their first session.

> **NOTE**
>
> An OEM app store can be a mobile app or a cloud app.

User Authentication

App stores can handle user authentication for the installed cloud apps. For example, users can log in after installing a cloud app using the app store. After that, the app store will save an authentication token for the cloud app in the local policy table. Then, the cloud app can retrieve the authentication token from the local policy table and use it to authenticate a user with the application. If desired, an optional parameter, `CloudAppVehicleID`, can be used to identify the head unit.

Setting and Getting Cloud App Properties

An OEM's app store can manage the properties of a specific cloud app by setting and getting its `CloudAppProperties`. This table summarizes the properties that are included in `CloudAppProperties`:

PARAMETER NAME	DESCRIPTION
appID	appID for the cloud app
nicknames	List of possible names for the cloud app. The cloud app will not be allowed to connect if its name is not contained in this list
enabled	If true, cloud app will be displayed on HMI
authToken	Used to authenticate the user, if the app requires user authentication
cloudTransportType	Specifies the connection type Core should use. Currently Core supports WS and WSS , but an OEM can implement their own transport adapter to handle different values
hybridAppPreference	Specifies the user preference to use the cloud app version, mobile app version, or whichever connects first when both are available
endpoint	Remote endpoint for websocket connections

NOTE

Only trusted app stores are allowed to set or get `CloudAppProperties` for other cloud apps.

Setting Cloud App Properties

App stores can set properties for a cloud app by sending a `SetCloudAppProperties` request to Core to store them in the local policy table. For example, in this piece of code, the app store can set the `authToken` to associate a user with a cloud app after the user logs in to the app by using the app store:

OBJC | SWIFT

Getting Cloud App Properties

To retrieve cloud properties for a specific cloud app from local policy table, app stores can send `GetCloudAppProperties` and specify the `appId` for that cloud app as in this example:

OBJC | SWIFT

GETTING THE CLOUD APP ICON

Cloud app developers don't need to add any code to download the app icon. The cloud app icon will be automatically downloaded from the url provided by the policy table and sent to Core to be later displayed on the HMI.

Getting the Authentication Token

When users install cloud apps from an OEM's app store, they may be asked to login to that cloud app using the app store. After logging in, app store can save the `authToken` in the local policy table to be used later by the cloud app for user authentication.
A cloud app can retrieve its `authToken` from local policy table after starting the RPC service. The `authToken` can be used later by the app to authenticate the user:

OBJC | SWIFT

Getting CloudAppVehicleID (Optional)

The `CloudAppVehicleID` is an optional parameter used by cloud apps to identify a head unit. The content of `CloudAppVehicleID` is up to the OEM's implementation. Possible values could be the VIN or a hashed VIN.
The `CloudAppVehicleID` value can be retrieved as part of the `GetVehicleData` RPC. To find out more about how to retrieve `CloudAppVehicleID`, check out the Retrieving Vehicle Data section.

Encryption

Some OEMs may want to encrypt messages passed between your SDL app and the head unit. If this is the case, when you submit your app to the OEM for review, they will ask you to add a security library to your SDL app. It is also possible to encrypt messages even if the OEM does not require encryption. In this case, you will have to work with the OEM to get a security library. This section will show you how to add the security library to your SDL app and configure optional encryption.

When Encryption is Needed

OEM Required Encrypted RPCs

OEMs may want to encrypt all or some of the RPCs being transmitted between your SDL app and SDL Core. The library will handle encrypting and decrypting RPCs that are required to be encrypted.

OEM Required Encrypted Video and Audio

OEMs may want to encrypt video and audio streaming. Information on how to set up encrypted video and audio streaming can be found in Video Streaming for Navigation Apps

> Introduction. The library will handle encrypting the video and audio data sent to the head unit.

Optional Encryption

You may want to encrypt some or all of the RPCs you send to the head unit even if the OEM does not require that they be protected. In that case you will have to manually configure the payload protection status of every RPC that you send. Please note that if you require that an RPC be encrypted but there is no security manager configured for the connected head unit, then the RPC will not be sent by the library.

> **NOTE**
>
> For optional encryption to work, you must work with each OEM to obtain their proprietary security library.

Creating the Encryption Configuration

Each OEM that supports SDL will have their own proprietary security library. You must add all required security libraries in the encryption configuration when you are configuring the SDL app.

OBJC | SWIFT

Getting the Encryption Status

Since it can take a few moments to set up the encryption manager, you must wait until you know that setup has completed before sending encrypted RPCs. If your RPC is sent before setup has completed, your RPC will not be sent. You can implement the `SDLServiceEncry`

`ptionDelegate`, which is set in `SDLEncryptionConfiguration`, to get updates to the encryption manager state.

OBJC | SWIFT

Setting Optional Encryption

If you want to encrypt a specific RPC, you must configure the payload protected status of the RPC before you send it to the head unit. In order to send RPCs with optional encryption you must call `startRPCEncryption` on the `sdlManager` to make sure the encryption manager gets started correctly. The best place to put `startRPCEncryption` is in the successful callback of `startWithReadyHandler`.

OBJC | SWIFT

Then, once you know the encryption manager has started successfully via encryption manager state updates to your `SDLServiceEncryptionDelegate` object, you can start to send encrypted RPCs by setting `payloadProtected` to `true`.

OBJC | SWIFT

Introduction

Mobile navigation allows map partners to easily display their maps as well as present visual and audio turn-by-turn prompts on the head unit.

Navigation apps have different behavior on the head unit than normal applications. The main differences are:

- Navigation apps don't use base screen templates. Their main view is the video stream sent from the device.

- Navigation apps can send audio via a binary stream. This will attenuate the current audio source and should be used for navigation commands.
- Navigation apps can receive touch events from the video stream.

Configuring a Module to Stream

In order to view the stream, you need a head unit to connect with that supports streaming. If this is a physical module created by an OEM, such as a Ford TDK, you may need special permissions from that OEM to test streaming. Physical modules often have strict permissions and/or encryption requirements to stream.

The alternative is to stream over TCP to open-source Core. For more details on setting up open-source Core and an HMI, see the Install and Run guide, and to set up video streaming for that Core and HMI, see the Audio and Video Streaming guide. We recommend using the built-in Generic_HMI server streaming instead of GStreamer socket or pipe streaming.

Configuring a Navigation App

The basic connection setup is similar for all apps. Please follow the Integration Basics guide for more information.

In order to create a navigation app an `appType` of `SDLAppHMITypeNavigation` must be set in the `SDLManager`'s `SDLLifecycleConfiguration`.

The second difference is that a `SDLStreamingMediaConfiguration` must be created and passed to the `SDLConfiguration`. A property called `securityManagers` must be set if connecting to a version of Core that requires secure video and audio streaming. This property requires an array of classes of security managers, which will conform to the `SDLSecurityType` protocol. These security libraries are provided by the OEMs themselves, and will only work for that OEM. There is no general catch-all security library.

OBJC | SWIFT

MUST

When compiling your app for production, make sure to include all possible OEM security managers that you wish to support.

Preventing Device Sleep

When building a navigation app, you should ensure that the device never sleeps while your app is in the foreground of the device and is in an HMI level other than `NONE`. If your device sleeps, it will be unable to stream video data. To do so, implement the following `SDLManagerDelegate` method.

OBJC | SWIFT

Keyboard Input

To present a keyboard (such as for searching for navigation destinations), you should use the `SDLScreenManager`'s keyboard presentation feature. For more information, see the Popup Keyboards guide.

Navigation Subscription Buttons

Head units supporting RPC v6.0+ may support navigation-specific subscription buttons for the navigation template. These subscription buttons allow your user to manipulate the map using hard buttons located on car's center console or steering wheel. It is important to support these subscription buttons in order to provide your user with the expected in-car navigation user experience. This is especially true on head units that don't support touch input as there will be no other way for your user to manipulate the map. See Template Subscription Buttons for a list of these navigation buttons.

When to Cancel Your Route

Between your navigation app, other navigation apps, and embedded navigation, only one route should be in progress at a time. To know when the embedded navigation or another navigation app has started a route, create a navigation service and when your service becomes inactive, your app should cancel any active route.

OBJC | SWIFT

Video Streaming (RPC v4.5+)

To stream video from a SDL app use the `SDLStreamingMediaManager` class. A reference to this class is available from the `SDLManager`. You can choose to create your own video streaming manager or you can use the `CarWindow` API to easily stream video to the head unit.

> **NOTE**
>
> Due to an iOS limitation, video can not be streamed when the app on the phone is in the background or the screen is off. Text will automatically be displayed telling the user that they must bring the application to the foreground. This text can be disabled by setting the `SDLStreamingMediaManager`'s `showVideoBackgroundDisplay` property to `false`.

Transports for Video Streaming

Transports are automatically handled for you. As of SDL v6.1+, the iOS library will automatically manage primary transports and secondary transports for video streaming. If

Wi-Fi is available, the app will automatically connect using it *after* connecting over USB / Bluetooth. This is the only way that Wi-Fi will be used in a production setting.

CarWindow

`CarWindow` is a system to automatically video stream a view controller screen to the head unit. When you set the view controller, `CarWindow` will resize the view controller's frame to match the head unit's screen dimensions. Then, when the video service setup has completed, it will capture the screen and send it to the head unit.

To start, you will have to set a `rootViewController`, which can easily be set using one of the convenience initializers: `autostreamingInsecureConfigurationWithInitialViewController:` or `autostreamingSecureConfigurationWithSecurityManagers:initialViewController:`

MUST

The view controller you are streaming must be a subclass of `SDLCarWindowViewController` or have only one `supportedInterfaceOrientation`. The `SDLCarWindowViewController` class prevents the `rootViewController` from rotating. This is necessary because rotation between landscape and portrait modes can cause the app to crash while the `CarWindow` API is capturing an image.

There are several customizations you can make to `CarWindow` to optimize it for your video streaming needs:

1. Choose how `CarWindow` captures and renders the screen using the `carWindowRenderingType` enum.
2. By default, when using `CarWindow`, the `SDLTouchManager` will sync its touch updates to the framerate. To disable this feature, set `SDLTouchManager.enableSyncedPanning` to `NO`.

3. As of SDL v7.1, if the HMI returns a desired framerate or max bitrate, the HMI's preferred settings will be use to configure the video encoder. You do have the option to change the default framerate and average bitrate via the `SDLStreamingMediaConfiguration.customVideoEncoderSettings`. Please note that your custom settings will override any settings received from the HMI except in the case where your custom framerate or average bitrate is larger than what the HMI says it can support.

Below are the video encoder defaults:

```
 @{
    __bridge NSString *)kVTCompressionPropertyKey_ProfileLevel: (__bridge NSString
*)kVTProfileLevel_H264_Baseline_AutoLevel,
    (__bridge NSString *)kVTCompressionPropertyKey_RealTime: @YES,
    (__bridge NSString *)kVTCompressionPropertyKey_ExpectedFrameRate: @15,
     __bridge NSString *)kVTCompressionPropertyKey_AverageBitRate: @600000
};
```

Showing a New View Controller

Simply update `sdlManager.streamManager.rootViewController` to the new view controller. This will also update the haptic parser.

Mirroring the Device Screen vs. Off-Screen UI

It is recommended that you use an off-screen view controller for your UI. This view controller will appear on-screen in the car, while remaining off-screen on the device. It is possible to mirror your device screen, however we strongly recommend against this course of action.

> **NOTE**
>
> If you are using off-screen rendering, it is recommended that your on-screen view controller not rotate. If it does, the lock screen will also rotate. Nothing will break in this case, but the UI won't look good if it rotates while your app is streaming.

OFF-SCREEN

To set an off-screen view controller all you have to do is instantiate a new `UIViewController` class and use it to set the `rootViewController`.

OBJC | SWIFT

MIRRORING THE DEVICE SCREEN

If you must use mirroring to stream video please be aware of the following limitations:

1. Getting the app's topmost view controller using `UIApplication.shared.keyWindow.rootViewController` will not work as this will give you SDL's lock screen view controller. The projected image you see in the car will be distorted because the view controller being projected will not be resized correctly. Instead, the `rootViewController` should be set in the `viewDidAppear:animated` method of the `UIViewController`.
2. If mirroring your device's screen, the `rootViewController` should only be set after `viewDidAppear:animated` is called. Setting the `rootViewController` in `viewDidLoad` or `viewWillAppear:animated` can cause weird behavior when setting the new frame.
3. If setting the `rootViewController` when the app returns to the foreground, the app should register for the `UIApplicationDidBecomeActive` notification and not the `UIApplicationWillEnterForeground` notification. Setting the frame after a notification from the latter can also cause weird behavior when setting the new frame.
4. Configure your SDL app so the lock screen is always visible. If you do not do this, video streaming can stop when the device is rotated.

Supporting Different Video Streaming View Sizes (SDL v7.1+, RPC v7.1+)

Some HMIs support multiple view sizes and may resize your SDL app's view during video streaming (i.e. to a collapsed view, split screen, preview mode or picture-in-picture). By default, your app will support all the view sizes and the `CarWindow` will resize the view controller's frame when the HMI notifies the app of the updated screen size. If you wish to support only some screen sizes, you can configure the `supportedPortraitStreamingRange` and `supportedLandscapeStreamingRange` properties via the `SDLStreamingMediaConfiguration` before starting the video stream. This will allow you to limit support to one or a combination of minimum/maximum resolutions, minimum diagonal, or minimum/maximum aspect ratios. If you want to support all possible landscape or portrait sizes you can simply set `nil` for the streaming range. If you wish to disable support for all possible landscape or portrait orientations you can disable the streaming range using the `SDLVideoStreamingRange.disabled` configuration.

CREATING THE VIDEO STREAMING RANGES

Below are some examples of how to configure a supported video streaming range:

OBJC | SWIFT

SETTING THE VIDEO STREAMING RANGES

Once you have configured a supported video streaming range, you can use it to set the `supportedPortraitStreamingRange` or `supportedLandscapeStreamingRange` properties when you are configuring the `SDLStreamingMediaConfiguration`.

OBJC | SWIFT

> **NOTE**
>
> If you disable both the `supportedLandscapeStreamingRange` and `supportedPortraitStreamingRange`, video will not stream.

GETTING THE UPDATED SCREEN SIZE

If the HMI resizes the view during streaming, the video stream will automatically restart with the new size. If desired, you can subscribe to screen size updates via the `SDLStreamingVideoDelegate`.

OBJC | SWIFT

Sending Raw Video Data

If you decide to send raw video data instead of relying on the `CarWindow` API to generate that video data from a view controller, you must maintain the lifecycle of the video stream as there are limitations to when video is allowed to stream. The app's HMI state on the head unit and the app's application state on the device determines whether video can stream. Due to an iOS limitation, video cannot be streamed when the app on the device is no longer in the foreground and/or the device is locked/sleeping.

The lifecycle of the video stream is maintained by the SDL library. The `SDLManager.streamingMediaManager` can be accessed once the `start` method of `SDLManager` is called. The `SDLStreamingMediaManager` automatically takes care of determining screen size and encoding to the correct video format.

> **NOTE**
>
> It is not recommended to alter the default video format and resolution behavior as it can result in distorted video or the video not showing up at all on the head unit. However, that option is available to you by implementing `SDLStreamingMediaConfiguration.dataSource`.

Sending Video Data

To check whether or not you can start sending data to the video stream, watch for the `SDLVideoStreamDidStartNotification`, `SDLVideoStreamDidStopNotification`, and `SDLVideoStreamSuspendedNotification` notifications. When you receive the start notification, start sending video data; stop when you receive the suspended or stop notifications. You will receive a video stream suspended notification when the app on the device is backgrounded. There are parallel start and stop notifications for audio streaming.

Video data must be provided to the `SDLStreamingMediaManager` as a `CVImageBufferRef` (Apple documentation here). Once the video stream has started, you will not see video appear until Core has received a few frames. Refer to the code sample below for an example of how to send a video frame:

OBJC | SWIFT

```
CVPixelBufferRef imageBuffer = <#Acquire Image Buffer#>;

if ([self.sdlManager.streamManager sendVideoData:imageBuffer] == NO) {
  NSLog(@"Could not send Video Data");
}
```

Best Practices

- A constant stream of map frames is not necessary to maintain an image on the screen. Because of this, we advise that a batch of frames are only sent on map movement or location movement. This will keep the application's memory consumption lower.

- For the best user experience, we recommend sending at least 15 frames per second.

Handling HMI Scaling (RPC v6.0+)

If the HMI scales the video stream, you will have to handle scaling the projected view, touches and haptic rectangles yourself (this is all handled for you behind the scenes in the `CarWindow` API). To find out if the HMI scales the video stream, you must for query and check the `SDLVideoStreamingCapability` for the `scale` property. Please check the Adaptive Interface Capabilities section for more information on how to query for this property using the system capability manager.

Audio Streaming

A navigation app can stream raw audio to the head unit. This audio data is played immediately. If audio is already playing, the current audio source will be attenuated and your audio will play. Raw audio must be played with the following parameters:

- **Format**: PCM
- **Sample Rate**: 16k
- **Number of Channels**: 1
- **Bits Per Second (BPS)**: 16 bits per sample / 2 bytes per sample

To stream audio from a SDL app, use the `SDLStreamingMediaManager` class. A reference to this class is available from the `SDLManager`'s `streamManager` property.

Audio Stream Manager

The `SDLAudioStreamManager` will help you to do on-the-fly transcoding and streaming of your files in mp3 or other formats, or prepare raw PCM data to be queued and played.

Starting the Audio Manager

Like the lifecycle of the video stream, the lifecycle of the audio stream is maintained by the SDL library, therefore, you do not need to start the audio stream if you've set a

streaming configuration when starting your SDLManager. When you receive the SDLAudioStreamDidStartNotification, you can begin streaming audio.

PLAYING FROM FILE

OBJC | SWIFT

PLAYING FROM DATA

OBJC | SWIFT

IMPLEMENTING THE DELEGATE

OBJC | SWIFT

Manually Sending Data

Once the audio stream is connected, data may be easily passed to the Head Unit. The function `sendAudioData:` provides us with whether or not the PCM Audio Data was successfully transferred to the Head Unit. If your app is in a state that it is unable to send audio data, this method will return a failure. If successful playback will begin immediately.

OBJC | SWIFT

Touch Input

Navigation applications support touch events like single taps, double-taps, panning, and pinch gestures. You can use the `SDLTouchManager` class to get touch events, or you can manage the touch events yourself by listening for the `SDLDidReceiveTouchEventNotification` notification.

> **NOTE**
>
> You must have a valid and approved `appId` from an OEM in order to receive touch events.

Using SDLTouchManager

`SDLTouchManager` has multiple callbacks that will ease the implementation of touch events. You can register for callbacks through the stream manager:

OBJC | SWIFT

> **NOTE**
>
> The view passed from the following callbacks are dependent on using the built-in focusable item manager to send haptic rects. See supporting haptic input "Automatic Focusable Rects" for more information.

The following callbacks are provided:

OBJC | SWIFT

NOTE

Points that are provided via these callbacks are in the head unit's coordinate space. This is likely to correspond to your own streaming coordinate space. You can retrieve the head unit dimensions from `SDLStreamingMediaManager.screenSize`.

Implementing onTouchEvent Yourself

If you want access to the raw touch data, you can subscribe to the touch event notifications. The notification will contain the following data:

TYPE

TOUCH TYPE	WHAT DOES THIS MEAN?
BEGIN	Sent for the first touch event of a touch.
MOVE	Sent if the touch moved.
END	Sent when the touch is lifted.
CANCEL	Sent when the touch is canceled (for example, if a dialog appeared over the touchable screen while the touch was in progress).

EVENT

TOUCH EVENT	WHAT DOES THIS MEAN?
touchEventId	Unique ID of the touch. Increases for multiple touches (0, 1, 2, ...).
timeStamp	Timestamp of the head unit time. Can be used to compare time passed between touches.
coord	X and Y coordinates in the head unit coordinate system. (0, 0) is the top left.

EXAMPLE

> **NOTE**
>
> Please note that if you are integrating an sdl_ios version less than v6.3, the following example code will not work. We recommend updating to the latest release version.

OBJC | SWIFT

Supporting Haptic Input (RPC v4.5+)

SDL now supports "haptic" input: input from something other than a touch screen. This could include trackpads, click-wheels, etc. These kinds of inputs work by knowing which views on the screen are touchable and focusing / highlighting on those areas when the user moves the trackpad or click wheel. When the user selects within a view, the center of that area will be "touched".

> **NOTE**
>
> Currently, there are no RPCs for knowing which view is highlighted, so your UI will have to remain static (i.e. you should not create a scrolling menu in your SDL app).

You will also need to implement touch input support in order to receive touches on the views. In addition, you must support the automatic focusable item manager in order to receive a touched `UIView` in the `SDLTouchManagerDelegate` in addition to the `CGPoint`.

Automatic Focusable Rectangles

SDL has support for automatically detecting focusable views within your UI and sending that data to the head unit. You will still need to tell SDL when your UI changes so that it can re-scan and detect the views to be sent.

In order to use the automatic focusable item locator, you must set the `UIWindow` of your streaming content on `SDLStreamingMediaConfiguration.window`. So long as the window is set, the focusable item locator will start running. Whenever your app UI updates, you will need to send a notification:

OBJC | SWIFT

NOTE

When your renderingType is `SDLCarWindowRenderingTypeLayer`, the `SDLDidUpdateProjectionView` notification should only be sent in the overridden `viewDidLayoutSubviews` method of your `rootViewController`. If you do not, your haptic rects may not update as you expect.

SDL can only automatically detect `UIButton`s and anything else that responds `true` to `canBecomeFocused`. This means that custom `UIView` objects will *not* be found. You must send these objects manually, see "Manual Focusable Rects".

Before Xcode 12.5, some built-in `UIView` subclasses, such as `UITextField`, responded `true` to `canBecomeFocused`. That is not longer true, and you must subclass these built-in views and implement `canBecomeFocused` to return `true`.

Manual Focusable Rects

If you need to supplement the automatic focusable item locator, or do all of the location yourself (e.g. views that are not focusable such as custom UIViews or OpenGL views), then you will have to manually send and update the focusable rects using `SDLSendHapticData`. This request, when sent replaces all current rects with new rects; so, if you want to clear all of the rects, you would send the RPC with an empty array. Or, if you want to add a single rect, you must re-send all previous rects in the same request.

Usage is simple, you create the rects using `SDLHapticRect`, add a unique id, and send all the rects using `SDLSendHapticData`.

OBJC | SWIFT

Displaying Turn Directions

While your app is navigating the user, you will also want to send turn by turn directions. This is useful for if your app is in the background or if the user is in the middle of a phone call, and gives the system additional information about the next maneuver the user must make.
When your navigation app is guiding the user to a specific destination, you can provide the user with visual and audio turn-by-turn prompts. These prompts will be presented even when your SDL app is backgrounded or a phone call is ongoing.
While your app is navigating the user, you will also want to send turn by turn directions. This is useful if your app is in the background or if the user is in the middle of a phone call, and gives the system additional information about the next maneuver the user must make.

To create a turn-by-turn direction that provides both a visual and audio cues, a combination of the `SDLShowConstantTBT` and `SDLAlertManeuver` RPCs must should be sent to the head unit.

> **NOTE**
>
> If the connected device has received a phone call in the vehicle, the `SDLAlertManeuver` is the only way for your app to inform the user of the next turn.

Visual Turn Directions

The visual data is sent using the `SDLShowConstantTBT` RPC. The main properties that should be set are `navigationText1`, `navigationText2`, and `turnIcon`. A best practice for navigation apps is to use the `navigationText1` as the direction to give (i.e. turn right) and `navigationText2` to provide the distance to that direction (i.e. 3 mi.).

Audio Turn Directions

The audio data is sent using the `SDLAlertManeuver` RPC. When sent, the head unit will speak the text you provide (e.g. In 3 miles turn right).

Sending Audio and Visual Turn Directions

OBJC | SWIFT

Remember when sending a `SDLImage`, that the image must first be uploaded to the head unit with the `SDLFileManager`.

Clearing the Turn Directions

To clear a navigation direction from the screen, send a `SDLShowConstantTBT` with the `maneuverComplete` property set to true. This will also clear the accompanying `SDLAlertManeuver`.

OBJC | SWIFT

Video Streaming Menu

When building a video-streaming navigation application, you can choose to create a custom menu using your own UI or use the built-in SDL menu system. The SDL menu allows you to display a menu structure so users can select menu options or submenus. For more information about the SDL menu system, see menus. It's recommended to use the built-in SDL menu system to have better performance, automatic driver distraction support - such as list limitations and text sizing, and more.

To open the SDL built-in menu from your video streaming UI, see 'Opening the Built-In Menu' below.

Opening the Built-In Menu

The Show Menu RPC allows you to open the menu programmatically. That way, you can open the menu from your own UI.

Show Top Level Menu

To show the top level menu use `sdlManager.screenManager.openMenu`.

OBJC | SWIFT

Show Sub-Menu

You can also open the menu directly to a sub-menu. This is further down the tree than the top-level menu. To open a sub-menu, pass a cell that contains sub-cells. If the cell has no sub-cells the method call will fail.

> **NOTE**
>
> The sub-cell you use in `openSubMenu` must be included in `sdlManager.screenManager.menu` array. If it is not included in the array, the method call will fail.

OBJC | SWIFT

Close Application

If you choose to not use the built-in SDL menu system and instead want to use your own menu UI, you need to have a way for users to close your application. This should be done through a menu option in your UI that sends the `CloseApplication` RPC.

> **NOTE**
>
> This RPC is unnecessary if you are using `OpenMenu` because OEMs will take care of providing a close button into your menu themselves.

OBJC | SWIFT

Configuring SDL Logging

A powerful built-in logging framework is available to make debugging your SDL app easier. It provides many of the features common to other 3rd party logging frameworks for iOS and can be used by your own app as well. We recommend that your app's integration with SDL provide logging using this framework rather than any other 3rd party framework your app may be using or `NSLog`. This will consolidate all SDL related logs in a common format and to common destinations.

SDL will configure its logging into a production-friendly configuration by default. If you wish to use a debug or a custom configuration, then you will have to specify this yourself. `SDLConfiguration` allows you to pass a `SDLLogConfiguration` with custom values. A few of these values will be covered in this section, the others are in their own sections below.

When setting up your `SDLConfiguration` you can pass a different log configuration:

OBJC | SWIFT

Format Type

Currently, SDL provides three output formats for logs (for example into the console or file log), these are "Simple", "Default", and "Detailed".

Simple:

```
09:52:07:324 ▯ (SDL)Protocol – I'm a log!
```

Default:

```
09:52:07:324 ▯ (SDL)Protocol:SDLV2ProtocolHeader:25 – I'm also a log!
```

Detailed:

```
09:52:07:324 ▯ DEBUG com.apple.main-thread:(SDL)Protocol:[SDLV2ProtocolHeader parse:]:74 – Me three!
```

Log Synchronicity

The configuration provides two properties, `asynchronous` and `errorsAsynchronous`. By default `asynchronous` is true and `errorsAsynchronous` is false. This means that any logs that are not logged at the error log level will be logged asynchronously on a separate serial queue, while those on the error log level will be logged synchronously on the separate queue (but the thread that logged it will be blocked until that log completes).

Log level

The `globalLogLevel` defines which logs will be logged to the target outputs. For example, if you set the log level to `debug`, all error, warning, and debug level logs will be logged,

but verbose level logs will not be logged.

SDLLOGLEVEL	VISIBLE LOGS
Off	none
Error	error
Warning	error and warning
Debug	error, warning and debug
Verbose	error, warning, debug and verbose

NOTE

Although the `default` log level is defined in the SDLLogLevel enum, it should not be used as a global log level. See the API documentation for more detail.

Targets

Targets are the output locations where the log will appear. By default only the OSLog log target will be enabled in both default and debug configurations. You may configure additional pre-built targets or create your own targets and add them.

APPLE SYSTEM LOG TARGET (DEPRECATED)

The Apple System Logger target, `SDLLogTargetAppleSystemLogger` is now deprecated in favor of the OS Log target which will do the same thing. It will be removed in a future release. This target will log to the Xcode console and the device console.

OS LOG TARGET

The OSLog target, `SDLLogTargetOSLog`, is the default log target in both default and debug configurations. For more information on this logging system see Apple's documentation. SDL's OSLog target will take advantage of subsystems and levels to allow you powerful runtime filtering capabilities through the MacOS Console app with a connected device.

FILE TARGET

The File target, `SDLLogTargetFile`, allows you to log messages to a rolling set of files which will be stored on the device, specifically in the `Documents/smartdevicelink/log/` folder. The file names will be timestamped with the start time.

To access the file, you can either access it from runtime on the device (for example, to attach it to an email that the user sends), or if you have access to the device, you can access them via iTunes (pre-Catalina) or the MacOS Finder (post-Catalina). To access the files on the device you must make the following small modifications to your app:

MACOS CATALINA OR LATER

1. Add the key `UIFileSharingEnabled` to your `info.plist`. Set the value to `YES`.
2. Connect the device to a MacOS computer.
3. Open the Finder, click on the device in the sidebar, then click on "Files" > "Your App Name".
4. You should see a folder called "smartdevicelink". Drag and drop the folder to your desktop (or somewhere in your file system). When you open the folder on your computer, you will see the log files for each session (default maxes out at 3).

MACOS PRE-CATALINA

1. Add the key `UIFileSharingEnabled` to your `info.plist`. Set the value to `YES`.

2. Connect the device to a computer that has iTunes installed.
3. Open iTunes, click on the icon for the device, then click on "File Sharing" > "Your App Name".
4. You should see a folder called "smartdevicelink". Select the folder and click "Save". When you open the folder on your computer, you will see the log files for each session (default maxes out at 3).

FILE LOGGING AND PRODUCTION RELEASES

1. You should remove the file sharing enabled info.plist key before submitting your app to Apple.
2. If you are testing an archive build, you will only be able to view error and warning logs if the build configuration was set to "release". To get debug and/or verbose logs you must create the archive build with the build configuration set to "debug".

CUSTOM LOG TARGETS

The protocol all log targets conform to, `SDLLogTarget`, is public. If you wish to make a custom log target in order to, for example, log to a server, it should be fairly easy to do so. If it can be used by other developers and is not specific to your app, then submit it back to the SmartDeviceLink iOS library project! If you want to add targets *in addition* to the default target that will output to the console:

OBJC | SWIFT

```
logConfig.targets = [logConfig.targets
setByAddingObjectsFromArray:@[[SDLLogTargetFile logger]]];
```

Modules

A module is a set of files packaged together. Create modules using the `SDLLogFileModule` class and add it to the configuration. Modules are used when outputting a log message. The log message may specify a module instead of a specific file name for clarity's sake. The SDL library will automatically add the modules corresponding to its own files after you submit your configuration. For your specific use case, you may wish to provide a module corresponding to your whole app's integration and simply name it with

your app's name, or, you could split it up further if desired. To add modules to the configuration:

OBJC SWIFT

```
logConfig.modules = [logConfig.modules
setByAddingObjectsFromArray:@[[SDLLogFileModule moduleWithName:@"Test"
files:[NSSet setWithArray:@[@"File1", @"File2"]]]]];
```

Filters

Filters are a compile-time concept of filtering in or out specific log messages based on a variety of possible factors. Call `SDLLogFilter` to easily set up one of the default filters or to create your own using a custom `SDLLogFilterBlock`. You can filter to only allow certain files or modules to log, only allow logs with a certain string contained in the message, or use regular expressions.

OBJC SWIFT

```
SDLLogFilter *filter = [SDLLogFilter filterByDisallowingString:@"Test"
caseSensitive:NO];
```

Logging with the SDL Logger

In addition to viewing the library logs, you also have the ability to log with the SDL logger. All messages logged through the SDL logger, including your own, will use your `SDLLogConfiguration` settings.

Objective-C Projects

First, import the `SDLLogMacros` header.

```
#import "SDLLogMacros.h"
```

Then, simply use the convenient log macros to create a custom SDL log in your project.

```
SDLLogV(@"This is a verbose log");
SDLLogD(@"This is a debug log");
SDLLogW(@"This is a warning log");
SDLLogE(@"This is an error log");
```

Swift Projects

To add custom SDL logs to your Swift project you must first install a submodule called **SmartDeviceLink/Swift**.

COCOAPODS

If the SDL iOS library was installed using CocoaPods, simply add the submodule to the **Podfile** and then install by running `pod install` in the root directory of the project.

```
target '<#Your Project Name#>' do
    pod 'SmartDeviceLink', '~> <#SDL Version#>'
    pod 'SmartDeviceLink/Swift', '~> <#SDL Version#>'
end
```

SWIFT PACKAGE MANAGER

If the SDL iOS library was installed using Swift Package Manager, install the `SmartDeviceLinkSwift` target to your SPM installation. Then, where you want to log, `import SmartDeviceLinkSwift`.

LOGGING IN SWIFT

Once you have access to the SmartDeviceLinkSwift enhancements, you can use the `SDLLog` functions in your project.

```
SDLLog.v("This is a verbose log")
SDLLog.d("This is a debug log")
SDLLog.w("This is a warning log")
SDLLog.e("This is an error log")
```

Updating from v6.7 to v7.0

The iOS library has made a number of breaking changes in SDL v7.0. This means that your project is unlikely to compile without changes.

iOS Minimum Version Changes

SDL iOS 7.0 now requires that your app's minimum supported version be iOS 10.0 or greater – previously it was iOS 8.0. If your app's minimum version is already iOS 10.0 or greater, then there's nothing you need to do! However, if you target a lower iOS version as your minimum version, you will need to stay on SDL iOS v6.7 until you can move up your minimum version. SDL iOS 7.0 has removed functionality shims necessary to allow it to function properly on iOS versions below 10.0.

Changes to RPC Initializers

All deprecated methods have been removed. Most of the removed methods are RPC initializers. If you are affected by this change, please look at the new available initializers and choose one of those to use.

For example:

```
// This was deprecated and now removed
SDLAlert *alert = [[SDLAlert alloc] initWithAlertText1:@"Text1" alertText2:@"Text2"
alertText3:@"Text3"];

// Replacement
SDLAlert *alert = [[SDLAlert alloc] init];
alert.alertText1 = @"Text1";
alert.alertText2 = @"Text2";
alert.alertText3 = @"Text3";
```

We will be looking to improve RPC initializers in the future.

Other API Removals

CONFIGURATIONS

Previously deprecated `SDLConfiguration` and other configuration APIs have been removed.

In `SDLLifecycleConfiguration`, the `SDLLifecycleConfiguration defaultConfigurationWithAppName:appId:` method and it's `debugConfiguration` counterpart have been officially removed. You must now use `defaultConfigurationWithAppName:fullAppId:` and it's `debugConfiguration` counterpart. Note that if you set a legacy app id into the `fullAppId` field, everything will continue to work as it did on the previous API.

Several `SDLStreamingMediaConfiguration` APIs have also been removed. Any API that took a security manager is now gone. In their stead, add your security manager onto an `SDLEncryptionConfiguration` instance and pass it to the `SDLConfiguration`.

DELEGATE API REMOVALS

Delegate API removals require special attention because if you still implement the deprecated method, that method will no longer work and there will no longer be a warning.

`SDLKeyboardDelegate updateAutocompleteWithInput:completionHandler:` has been removed and is superseded by `updateAutocompleteWithInput:autoCompleteResultsHandler:`. The new method allows you to return a list of results. On older head units, only the top result will be used.

`SDLManagerDelegate managerShouldUpdateLifecycleToLanguage:` has been removed and is superseded by `managerShouldUpdateLifecycleToLanguage:hmiLangugage:`. The new method will alert you if either the VR language or the text language changes.

PERMISSION MANAGER

The `SDLPermissionManager` has had several API removals. The primary change is to use the `SDLRPCFunctionName` enum in place of the `SDLPermissionRPCName` `NSString` typedef. This provides additional type safety when checking permissions for an RPC. Also note that `subscribeToRPCPermissions:groupType:withHandler:` has slightly different behavior than `addObserverForRPCs:groupType:withHandler:` when the `groupType` is `SDLPermissionGroupTypeAllAllowed` for the initial callback. It will now only callback if all items are allowed, whereas before it would callback no matter the initial status of the group.

Other Deprecations

We did also deprecate a few APIs in this release.

1. `SDLServiceUpdateReason` enums were not correctly formed. We deprecated the previous APIs and introduced new ones that are correctly formed.
2. Existing `SDLCharacterSet` sets were not standards-compliant and are deprecated. New character sets have been added and will be used in future head units to describe text fields.
3. The `SDLLockScreenStatus` notification now has a new type of payload. It has changed from an `SDLOnLockScreenStatus` RPC to a `SDLLockScreenStatusInfo` object. This is only important if you have built your own lock screen management system instead of using the one provided through the SDL iOS library.

New Features

CHANGING TEMPLATE LAYOUT

The primary new feature is `SDLManager.screenManager changeLayout:withCompletionHandler:`. This wraps the `SDLSetDisplayLayout` and `SDLShow` (on RPC v6.0+) ability to change the template layout and color scheme. `SDLSetDisplayLayout` will be deprecated in a future release, and this is now the preferred API to manage layouts.

Can I Integrate SDL into a React Native App?

SDL does work and can be integrated into a React Native application.

Please follow the React Native Getting Started guide for how to create a new React Native application if you need one. To install SDL into your React Native app, you will need to follow the React Native Native Module's guide to integrate the SDL library into your application using React Native's Native Modules feature. You must make sure you have Native Modules installed as a dependency in order to use 3rd party APIs in a React Native application. If this is not done your app will not work with SmartDeviceLink. Native API methods are not exposed to JavaScript automatically, this must be done manually by you. Then see the SDL Installation Guide for more information on installing SDL's native library.

NOTE

This guide is not meant to walk you through how to make a React Native app but help you integrate SDL into an existing application. We will show you a basic example of how to communicate between your app's JavaScript code and SDL's native Obj-C code. For more advanced features, please refer to the React Native documentation linked above.

Integration Basics

Native API methods are not exposed automatically to JavaScript. This means you must expose methods you wish to use from SDL to your React Native app. You must implement the `RCTBridgeModule` protocol into a bridge class (see below for an example). Please follow SmartDeviceLink Integration Basics guide for the basic setup of a native SDL `ProxyManager` class that your bridge code will communicate with. This is the necessary starting point in order to continue with this example. Also set up a simple UI with buttons and some text on the SDL side.

Creating the RCTBridge

To create a native module you must implement the `RCTBridgeModule` protocol. Update your `ProxyManager` to include `RCTBridgeModule`.

OBJECTIVE-C

ProxyManager.h

```
#import <React/RCTBridgeModule.h>

@interface ProxyManager : NSObject <RCTBridgeModule>

<#Proxy Manager code#>

@end
```

ProxyManager.m

An `RCT_EXPORT_MODULE()` macro must be added to the implementation file to expose the class to React Native.

```
@implementation ProxyManager

RCT_EXPORT_MODULE();
<#Proxy Manager code#>

@end
```

SWIFT

Before you move forward, you must add `#import "React/RCTBridgeModule.h"` to your `Bridging Header`. When creating a Swift application and importing Objective-C code, Xcode should ask if it should create this header file for you. You can create this file manually as well. You must include this bridging header for your React Native app to work.

```
@objc(ProxyManager)
class ProxyManager: NSObject {
<#Proxy Manager Code#>
}
```

Next, to expose the above Swift class to React Native, you must create an Objective-C file and wrap the Swift class name in a `RCT_EXTERN_MODULE` in order to use the Swift class in a React Native app.
ProxyManager.m

```
#import "React/RCTBridgeModule.h"
@interface RCT_EXTERN_MODULE(ProxyManager, NSObject)
@end
```

Emitting Event Notifications to JavaScript

Inside the `ProxyManger` class, post a notification for a particular event you wish to execute. The 'Event Emitter' class, which you will see later in the documentation, will observe this event notification and will call the React Native listener that you will set up later in the documentation below.

Inside the `ProxyManager` add a soft button to your SDL HMI. Inside the soft button handler, post the notification and pass along a reference to the `sdlManager` in order to

update your React Native UI through the bridge.

OBJC SWIFT

```
SDLSoftButtonObject *softButton = [[SDLSoftButtonObject alloc]
initWithName:@"Button" state:[[SDLSoftButtonState alloc]
initWithStateName:@"State 1" text:@"Data" artwork:nil] handler:^(SDLOnButtonPress
* _Nullable buttonPress, SDLOnButtonEvent * _Nullable buttonEvent) {
    if (buttonPress == nil) { return; }

    NSDictionary *userInfo = @{@"sdlManager": self.sdlManager};
    [[NSNotificationCenter defaultCenter] postNotificationName:<#Notification
Name#> object:nil userInfo:managers];
}];

self.sdlManager.screenManager.softButtonObjects = @[softButton];
```

CREATE THE EVENTEMITTER BRIDGE CLASS

Create the class that will be the listener for the notification you created above. This class will be sending and receiving messages from your JavaScript code (React Native). The required `supportedEvents` method returns an array of supported event names. Sending an event name that is not included in the array will result in an error. An "event" is sending a message from native code to React Native code.

OBJECTIVE-C

SDLEventEmitter.h

```
#import <React/RCTEventEmitter.h>
#import <React/RCTBridgeModule.h>
#import <Foundation/Foundation.h>

NS_ASSUME_NONNULL_BEGIN

@interface SDLEventEmitter : RCTEventEmitter

@end

NS_ASSUME_NONNULL_END
```

SDLEventEmitter.m

```
#import "SDLEventEmitter.h"
#import "ProxyManager.h"
#import <React/RCTConvert.h>
#import <SmartDeviceLink/SmartDeviceLink.h>

@implementation SDLEventEmitter

RCT_EXPORT_MODULE()

- (instancetype)init {
    self = [super init];
    // Subscribe to event notifications sent from ProxyManager
    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(getDoActionNotification:) name:<#Notification Name#>
object:nil];

    return self;
}

// Required Method defining known action names
- (NSArray<NSString *> *)supportedEvents {
    return @[@"DoAction"];
}

// Run this code when the subscribed event notification is received
- (void)getDoActionNotification:(NSNotification *)notification {
    if(self.sdlManager == nil) {
        self.sdlManager = notification.userInfo[@"sdlManager"];
    }

    // Send the event to your React Native code with a dictionary of information
    [self sendEventWithName:@"DoAction" body:@{@"type": @"actionType"}];
}

@end
```

SWIFT

```swift
@objc(SDLEventEmitter)
class SDLEventEmitter: RCTEventEmitter {

    override init() {
        // Subscribe to event notifications sent from ProxyManager
        NotificationCenter.default.addObserver(self, selector: #selector(doAction(_:)),
name: Notification.Name(rawValue: "<#Notification Name#>", object: nil)
        super.init()
    }

    // Required Method defining known action names
    override func supportedEvents() -> [String]! {
        return ["DoAction"]
    }

    // Run this code when the subscribed event notification is received
    @objc func doAction(_ notification: Notification) {
        if self.sdlManger == nil {
            self.sdlManager = notification.userInfo["sdlManager"]
        }

        // Send the event to your React Native code with a dictionary of information
        sendEvent(withName: "DoAction", body: ["type": "actionType"])
    }

}
```

JAVASCRIPT

The above example will call into your JavaScript code with an event type `DoAction`. Inside your React Native (JavaScript) code, create a `NativeEventEmitter` object within your `EventEmitter` module and add a listener for the event.

```
import { NativeEventEmitter, NativeModules } from 'react-native';
const  { SDLEventEmitter } = NativeModules;

const testEventEmitter = new NativeEventEmitter(SDLEventEmitter);

// Build a listener to listen for events
const testData = testEventEmitter.addListener(
    'DoAction',
        () => SDLEventEmitter.eventCall({
            "data": {
                "low": "77",
                "high": "87",
                "currentTemp": "82",
                "rain": "50%"
            }
        }
    )
)
```

Exposing Native Methods to JavaScript

The last step is to wrap any native code methods you wish to expose to your JavaScript code inside `RCT_EXPORT_METHOD` for Objective-C and `RCT_EXTERN_METHOD` for Swift. We've seen above how native code can send notifications to your JavaScript code, now we will see how your JavaScript code can send notifications into your native SmartDeviceLink code. Inside the `SDLEventEmitter.m` file add the following method:

OBJECTIVE-C

```
RCT_EXPORT_METHOD(eventCall:(NSDictionary *)dict) {
    [self.sdlManager.screenManager beginUpdates];

    self.sdlManager.screenManager.textField1 = [NSString stringWithFormat:@"Low: %@ °F", [RCTConvert NSString:dict[@"data"][@"low"]]];
    self.sdlManager.screenManager.textField2 = [NSString stringWithFormat:@"High: %@ °F", [RCTConvert NSString:dict[@"data"][@"high"]]];

    [self.sdlManager.screenManager endUpdatesWithCompletionHandler:^(NSError * _Nullable error) {
        if (error != nil) {
            <#Error#>
        } else {
            <#Success#>
        }
    }];
}
```

SWIFT

If you're making a React Native application and using native Swift code, you will need to create the Objective-C bridger for the `SDLEventEmitter` class you created above. Wrap the method(s) you wish to expose in a `RCT_EXTERN_METHOD` macro inside your wrapper class. This wrapper will allow the JavaScript code to talk with your native code.

> **NOTE**
>
> Make sure you add `#import "React/RCTEventEmitter.h"` to the apps bridging header.

```
#import "React/RCTBridgeModule.h"
#import "React/RCTEventEmitter.h"

@interface RCT_EXTERN_MODULE(SDLEventEmitter, RCTEventEmitter)

RCT_EXTERN_METHOD(eventCall:(eventCall: (id)dict))

@end
```

Add the following method to `SDLEventEmitter.swift`:

```
@objc func eventCall(_ dict: NSDictionary) {
    self.sdlManager.screenManager.beginUpdates()
    let data = dict["data"]! as! NSDictionary
    self.sdlManager.screenManager.textField1 = "Low: \(data["low"]!) °F")"
    self.sdlManager.screenManager.textField2 = "High: \(data["high"]!) °F")"
    self.sdlManager.screenManager.endUpdates()
}
```

By now you should have a basic React Native application that can send a message from the Native side to the React Native layer. If done correctly the application should update the SDL UI when clicking the soft button on the head unit. The above documentation walked you through how to send a message to React Native and receive a message containing data back.