sdl

iOS Documentation

Document current as of 11/20/2018 03:14 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.

Install SDL SDK

There are three different ways to install the SDL SDK in your project: CocoaPods, Carthage, 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
```

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. Use a script such as Carthage's to accomplish this.

SDK Configuration

1. Connect to a Remote System

If you do not have an SDL enabled head unit for testing, Manticore may work for you. Manticore is a web-based emulator for testing how your app reacts to real-world vehicle events, on-screen interactions, and voice recognition—just like it would in a vehicle.

You can also build the sdl_core project on an Ubuntu VM or computer. The sdl_core project is an emulator that lets you simulate sending and receiving remote procedure calls between a smartphone app and a SDL Core.

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>
```

4. 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. create 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.

Integration Basics

How SDL Works

SmartDeviceLink works by sending remote procedure calls (RPCs) back and forth between a smartphone application and the SDL Core. These RPCs allow you to build the user interface, detect button presses, play audio, and get vehicle data, among other things. You will use the SDL library to build your app on the SDL Core.

Set Up a Proxy Manager Class

You will need a class that manages the RPCs sent back and forth 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.

OBJECTIVE-C

ProxyManager.h

```
#import <Foundation/Foundation.h>

NS_ASSUME_NONNULL_BEGIN

@interface ProxyManager : NSObject

+ (instancetype)sharedManager;

@end

NS_ASSUME_NONNULL_END
```

ProxyManager.m

```
#import "ProxyManager.h"

NS_ASSUME_NONNULL_BEGIN

@interface ProxyManager ()

@end

@implementation ProxyManager

+ (instancetype)sharedManager {
    static ProxyManager* sharedManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedManager = [[ProxyManager alloc] init];
    });

    return sharedManager;
}

- (instancetype)init {
    self = [super init];
    if (!self) {
      return nil;
    }
}

@end

NS_ASSUME_NONNULL_END
```

SWIFT

```
class ProxyManager: NSObject {
  // Singleton
  static let sharedManager = ProxyManager()

  private override init() {
    super.init()
  }
}
```

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.

OBJECTIVE-C

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  // Initialize and start the proxy
  [[ProxyManager sharedManager] connect];
}

@end
```

SWIFT

```
class AppDelegate: UIResponder, UIApplicationDelegate {
  func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [
UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // Initialize and start the proxy
    ProxyManager.sharedManager.connect()

    return true
  }
}
```

Importing the SDL Library

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

OBJECTIVE-C

```
#import <SmartDeviceLink/SmartDeviceLink.h>
```

SWIFT

```
import SmartDeviceLink
```

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.

OBJECTIVE-C

```objc
#import "ProxyManager.h"
#import "SmartDeviceLink.h"

NS_ASSUME_NONNULL_BEGIN

@interface ProxyManager ()

@property (nonatomic, strong) SDLManager *sdlManager;

@end

@implementation ProxyManager

+ (instancetype)sharedManager {
    static ProxyManager *sharedManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedManager = [[ProxyManager alloc] init];
    });

    return sharedManager;
}

- (instancetype)init {
    self = [super init];
    if (!self) {
      return nil;
    }

    return self
}

@end

NS_ASSUME_NONNULL_END
```

SWIFT

```swift
class ProxyManager: NSObject {
  // Manager
  fileprivate var sdlManager: SDLManager!

  // Singleton
  static let sharedManager = ProxyManager()

  private override init() {
    super.init()
  }
}
```

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

OBJECTIVE-C

```
SDLLifecycleConfiguration* lifecycleConfiguration = [
SDLLifecycleConfiguration defaultConfigurationWithAppName:
@"<#App Name#>" fullAppId:@"<#App Id#>"];
```

SWIFT

```
let lifecycleConfiguration = SDLLifecycleConfiguration(appName:
"<#App Name#>", fullAppId: "<#App Id#>")
```

TCP

OBJECTIVE-C

```
SDLLifecycleConfiguration* lifecycleConfiguration = [
SDLLifecycleConfiguration debugConfigurationWithAppName:
@"<#App Name#>" fullAppId:@"<#App Id#>" ipAddress:@"<#IP
Address#>" port:<#Port#>];
```

SWIFT

```
let lifecycleConfiguration = SDLLifecycleConfiguration(appName:
"<#App Name#>", fullAppId: "<#App Id#>", ipAddress: "<#IP
Address#>", port: <#Port#>))
```

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.

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.

OBJECTIVE-C

```
lifecycleConfiguration.shortAppName = @"<#Shortened App Name#>"
;
```

SWIFT

```
lifecycleConfiguration.shortAppName = "<#Shortened App Name#>"
```

3. App Icon

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

OBJECTIVE-C

```
UIImage* appImage = [UIImage imageNamed:@"<#AppIcon Name#>"
];
if (appImage) {
  SDLArtwork* appIcon = [SDLArtwork persistentArtworkWithImage:
appImage name:@"<#Name to Upload As#>" asImageFormat:
SDLArtworkImageFormatPNG /* or SDLArtworkImageFormatJPG */];
  lifecycleConfiguration.appIcon = appIcon;
}
```

SWIFT

```
if let appImage = UIImage(named: "<#AppIcon Name#>") {
  let appIcon = SDLArtwork(image: appImage, name: "<#Name to
Upload As#>", persistent: true, as: .JPG /* or .PNG */)
  lifecycleConfiguration.appIcon = appIcon
}
```

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 SYNC3 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.

OBJECTIVE-C

```
lifecycleConfiguration.appType = SDLAppHMITypeMedia;
```

SWIFT

```
lifecycleConfiguration.appType = .media
```

5. Template Coloring

You can alter the appearance of your app on a head unit in a consistent way using template coloring APIs.

> NOTE
>
> This will only work when connected to head units running SDL Core v5.0 or later.

OBJECTIVE-C

```
SDLRGBColor *green = [[SDLRGBColor alloc] initWithRed:126 green:188
 blue:121];
SDLRGBColor *white = [[SDLRGBColor alloc] initWithRed:249 green:251
 blue:254];
SDLRGBColor *darkGrey = [[SDLRGBColor alloc] initWithRed:57 green:
78 blue:96];
SDLRGBColor *grey = [[SDLRGBColor alloc] initWithRed:186 green:198
blue:210];
lifecycleConfiguration.dayColorScheme = [[SDLTemplateColorScheme
alloc] initWithPrimaryRGBColor:green secondaryRGBColor:grey
backgroundRGBColor:white];
lifecycleConfiguration.nightColorScheme = [[SDLTemplateColorScheme
alloc] initWithPrimaryRGBColor:green secondaryRGBColor:grey
backgroundRGBColor:darkGrey];
```

SWIFT

```
let green = SDLRGBColor(red: 126, green: 188, blue: 121)
let white = SDLRGBColor(red: 249, green: 251, blue: 254)
let grey = SDLRGBColor(red: 186, green: 198, blue: 210)
let darkGrey = SDLRGBColor(red: 57, green: 78, blue: 96)
lifecycleConfiguration.dayColorScheme = SDLTemplateColorScheme(
primaryRGBColor: green, secondaryRGBColor: grey,
backgroundRGBColor: white)
lifecycleConfiguration.nightColorScheme = SDLTemplateColorScheme(
primaryRGBColor: green, secondaryRGBColor: grey,
backgroundRGBColor: darkGrey)
```

NOTE

You may only change the template coloring in the `lifecycleConfiguration` and in `SetDisplayLayout` RPC requests. You may only change the template coloring once per template. i.e. You cannot change to the same template you are already on using `SetDisplayLayout` and expect the coloring to change.

6. 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`.

OBJECTIVE-C

```
[SDLLockScreenConfiguration enabledConfiguration]
```

SWIFT

```
SDLLockScreenConfiguration.enabled()
```

7. 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.

OBJECTIVE-C

```
[SDLLogConfiguration defaultConfiguration]
```

SWIFT

```
SDLLogConfiguration.default()
```

8. 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.

OBJECTIVE-C

```
SDLConfiguration* configuration = [SDLConfiguration
configurationWithLifecycle:lifecycleConfiguration lockScreen:[
SDLLockScreenConfiguration enabledConfiguration] logging:[
SDLLogConfiguration defaultConfiguration] fileManager:[
SDLFileManagerConfiguration defaultConfiguration]];
```

SWIFT

```
let configuration = SDLConfiguration(lifecycle: lifecycleConfiguration,
lockScreen: .enabled(), logging: .default(), fileManager: .default())
```

9. Create a SDLManager

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

OBJECTIVE-C

```
self.sdlManager = [[SDLManager alloc] initWithConfiguration:
configuration delegate:self];
```

SWIFT

```
sdlManager = SDLManager(configuration: configuration, delegate: self)
```

10. 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`.

OBJECTIVE-C

```
- (void)connect {
    [self.sdlManager startWithReadyHandler:^(BOOL success, NSError *
_Nullable error) {
        if (success) {
            // Your app has successfully connected with the SDL Core
        }
    }];
}
```

SWIFT

```
func connect() {
    // Start watching for a connection with a SDL Core
    sdlManager.start { (success, error) in
        if success {
            // Your app has successfully connected with the SDL Core
        }
    }
}
```

> 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.

Example Implementation of a Proxy Class

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

OBJECTIVE-C

ProxyManager.h

```
#import <Foundation/Foundation.h>

NS_ASSUME_NONNULL_BEGIN

@interface ProxyManager : NSObject

+ (instancetype)sharedManager;
- (void)start;

@end

NS_ASSUME_NONNULL_END
```

ProxyManager.m

```
#import <SmartDeviceLink/SmartDeviceLink.h>

NS_ASSUME_NONNULL_BEGIN

static NSString* const AppName = @"<#App Name#>";
static NSString* const AppId = @"<#App Id#>";
@interface ProxyManager () <SDLManagerDelegate>

@property (nonatomic, strong) SDLManager* sdlManager;

@end

@implementation ProxyManager

+ (instancetype)sharedManager {
    static ProxyManager *sharedManager = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedManager = [[ProxyManager alloc] init];
    });

    return sharedManager;
}

- (instancetype)init {
    self = [super init];
    if (!self) {
        return nil;
    }

    // Used for USB Connection
    SDLLifecycleConfiguration* lifecycleConfiguration = [
SDLLifecycleConfiguration defaultConfigurationWithAppName:AppName
 fullAppId:AppId];

    // Used for TCP/IP Connection
//    SDLLifecycleConfiguration* lifecycleConfiguration =
[SDLLifecycleConfiguration
debugConfigurationWithAppName:AppName fullAppId:AppId
ipAddress:@"<#IP Address#>" port:<#Port#>];

    UIImage* appImage = [UIImage imageNamed:@"<#AppIcon
Name#>"];
    if (appImage) {
        SDLArtwork* appIcon = [SDLArtwork persistentArtworkWithImage:
appImage name:@"<#Name to Upload As#>" asImageFormat:
SDLArtworkImageFormatJPG /* or SDLArtworkImageFormatPNG */];
        lifecycleConfiguration.appIcon = appIcon;
    }
```

```
    lifecycleConfiguration.shortAppName = @"<#Shortened App
Name#>";
    lifecycleConfiguration.appType = [SDLAppHMIType MEDIA];

    SDLConfiguration* configuration = [SDLConfiguration
configurationWithLifecycle:lifecycleConfiguration lockScreen:[
SDLLockScreenConfiguration enabledConfiguration] logging:[
SDLLogConfiguration defaultConfiguration] fileManager:[
SDLFileManager defaultConfiguration]];

    self.sdlManager = [[SDLManager alloc] initWithConfiguration:
configuration delegate:self];

    return self;
}

- (void)connect {
    [self.sdlManager startWithReadyHandler:^(BOOL success, NSError *
_Nullable error) {
        if (success) {
            // Your app has successfully connected with the SDL Core
        }
    }];
}

#pragma mark SDLManagerDelegate
- (void)managerDidDisconnect {
    NSLog(@"Manager disconnected!");
}

- (void)hmiLevel:(SDLHMILevel *)oldLevel didChangeToLevel:(
SDLHMILevel *)newLevel {
    NSLog(@"Went from HMI level %@ to HMI Level %@", oldLevel,
newLevel);
}

@end

NS_ASSUME_NONNULL_END
```

```
import SmartDeviceLink

class ProxyManager: NSObject {
    private let appName = "<#App Name#>"
    private let appId = "<#App Id#>"

    // Manager
    fileprivate var sdlManager: SDLManager!

    // Singleton
    static let sharedManager = ProxyManager()

    private override init() {
        super.init()

        // Used for USB Connection
        let lifecycleConfiguration = SDLLifecycleConfiguration(appName:
appName, fullAppId: appId)

        // Used for TCP/IP Connection
        // let lifecycleConfiguration = SDLLifecycleConfiguration
(appName: appName, fullAppId: appId, ipAddress: "<#IP Address#>",
port: <#Port#>)

        // App icon image
        if let appImage = UIImage(named: "<#AppIcon Name#>") {
            let appIcon = SDLArtwork(image: appImage, name: "<#Name
to Upload As#>", persistent: true, as: .JPG /* or .PNG */)
            lifecycleConfiguration.appIcon = appIcon
        }

        lifecycleConfiguration.shortAppName = "<#Shortened App
Name#>"
        lifecycleConfiguration.appType = .media

        let configuration = SDLConfiguration(lifecycle:
lifecycleConfiguration, lockScreen: .enabled(), logging: .default(),
fileManager: .default())

        sdlManager = SDLManager(configuration: configuration, delegate:
self)
    }

    func connect() {
        // Start watching for a connection with a SDL Core
        sdlManager.start { (success, error) in
            if success {
                // Your app has successfully connected with the SDL Core
            }
```

```
        }
    }
}

//MARK: SDLManagerDelegate
extension ProxyManager: SDLManagerDelegate {
  func managerDidDisconnect() {
    print("Manager disconnected!")
  }

  func hmiLevel(_ oldLevel: SDLHMILevel, didChangeToLevel newLevel:
SDLHMILevel) {
    print("Went from HMI level \(oldLevel) to HMI level \(newLevel)")
  }
}
```

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 three 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. `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.

3. `managerShouldUpdateLifecycleToLanguage:` Called when the head unit 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.

Where to Go From Here

You should now be able to connect to a head unit or emulator. From here, learn about designing your main interface. For further details on connecting, see Connecting to a SDL Core.

Connecting to an Infotainment System

To connect to an emulator, such as Manticore or a local Ubuntu SDL Core-based emulator, make sure to implement a TCP (`debug`) connection. The emulator and app should be on the same network (i.e. remember to set the correct IP address and port number in the `SDLLifecycleConfiguration`). The IP will most likely be the IP address of the operating system running the emulator. The port will most likely be `12345` .

NOTE

Known issues due to using a TCP connection:

- When app is in the background mode, the app will be unable to communicate with SDL Core. This will work on IAP connections.
- Audio will not play on the emulator. Only IAP connections are currently able to play audio because this happens over the standard Bluetooth / USB system audio channel.
- You cannot connect to an emulator using a USB connection due to Apple limitations with IAP connections.

Connecting with a Vehicle Head Unit or a Development Kit (TDK)

Production

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

Debugging

If you are testing with a vehicle head unit or TDK and wish to see debug logs in Xcode while the app is running, you must either use another app called the relay app to help you connect to the device, or you may use Xcode 9 / iOS 11 wireless debugging. When using the relay app, make sure to implement a TCP connection, if using iOS 11 wireless debugging, implement a IAP connection. Please see the guide for the relay app to learn how to set up the connection between the device, the relay app and your app.

NOTE

The same issues apply when connecting the relay app with a TDK or head unit as do when connecting to SDL Core. Please see the issues above, under the *Connect with an Emulator* heading.

Adding the Lock Screen

The lock screen is a vital part of SmartDeviceLink, as the lock screen prevents the user from using your application while the vehicle is in motion. SDL takes care of the lock screen for you. It still allows you to use your own view controller if you prefer your own look, but still want the recommended logic that SDL provides for free.

A benefit to using the provided Lock Screen is that we also handle retrieving a lock screen icon for versions of Core that support it, so that you do not have to be concerned with what car manufacturer you are connected to. If you subclass the provided lock screen, you can do the same in your own lock screen view controller.

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`:

OBJECTIVE-C

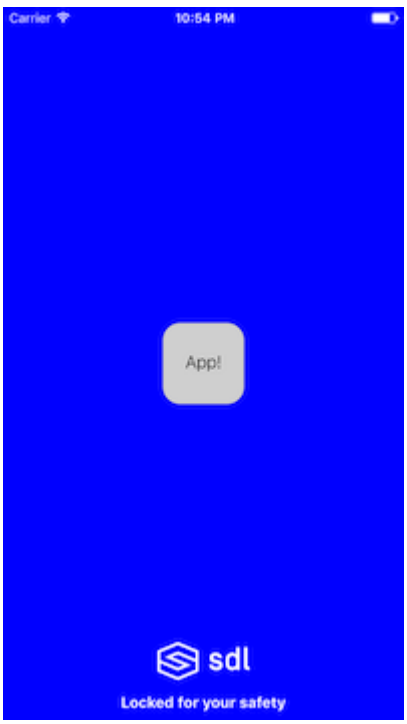
```
SDLLockScreenConfiguration *lockScreenConfiguration = [
SDLLockScreenConfiguration enabledConfiguration];
```

SWIFT

```
let lockScreenConfiguration = SDLLockScreenConfiguration.enabled()
```

Customizing the Provided Lock Screen

If you would like to use the provided lock screen but would like to add your own appearance to it, we provide that as well. `SDLLockScreenConfiguration` allows you to customize the background color as well as your app's icon. If the app icon is not included, we will use the SDL logo.



OBJECTIVE-C

```
UIImage *appIcon = <# Retreive App Icon #>
UIColor *backgroundColor = <# Desired Background Color #>
SDLLockScreenConfiguration *lockScreenConfiguration = [
SDLLockScreenConfiguration enabledConfigurationWithAppIcon:appIcon
 backgroundColor:backgroundColor];
```

SWIFT

```
let appIcon: UIImage = <# Retrieve App Icon #>
let backgroundColor: UIColor = <# Desired Background Color #>
let lockScreenConfiguration = SDLLockScreenConfiguration.
enabledConfiguration(withAppIcon: appIcon, backgroundColor:
backgroundColor)
```

Using Your Own Lock Screen

If you would like to use your own lock screen instead of the provided SDL one, but still use the logic we provide, you can use a new initializer within `SDLLockScreenConfiguration`:

OBJECTIVE-C

```
UIViewController *lockScreenViewController = <# Initialize Your View
Controller #>;
SDLLockScreenConfiguration *lockScreenConfiguration = [
SDLLockScreenConfiguration enabledConfigurationWithViewController:
lockScreenViewController];
```

SWIFT

```
let lockScreenViewController = <# Initialize Your View Controller #>
let lockScreenConfiguration = SDLLockScreenConfiguration.
enabledConfiguration(with: lockScreenViewController)
```

Using the Vehicle's Icon

If you want to build your own lock screen view controller, it is recommended that you subclass `SDLLockScreenViewController` and use the public `appIcon`, `vehicleIcon`, and `backgroundColor` properties.

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 be 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 `RegisterAppInterfaceResponse` 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 recogize 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.

OBJECTIVE-C

```
SDLLanguage headUnitLanguage = self.sdlManager.registerResponse.
language;
SDLLanguage headUnitHMIDisplayLanguage = self.sdlManager.
registerResponse.hmiDisplayLanguage;
```

SWIFT

```
let headUnitLanguage = sdlManager.registerResponse?.language
let headUnitHMIDisplayLanguage = sdlManager.registerResponse?.
hmiDisplayLanguage
```

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:` method. If the head unit's language is different from the default language and is a supported language, the method will be called with the head unit's current language. Return a `SDLLifecycleConfigurationUpdate` object with the new `appName` and/or `ttsName`.

OBJECTIVE-C

```
- (nullable SDLLifecycleConfigurationUpdate *)
managerShouldUpdateLifecycleToLanguage:(SDLLanguage)language {
    SDLLifecycleConfigurationUpdate *configurationUpdate = [[
SDLLifecycleConfigurationUpdate alloc] init];

    if ([language isEqualToEnum:SDLLanguageEnUs]) {
        update.appName = <#App Name in English#>;
    } else if ([language isEqualToEnum:SDLLanguageEsMx]) {
        update.appName = <#App Name in Spanish#>;
    } else if ([language isEqualToEnum:SDLLanguageFrCa]) {
        update.appName = <#App Name in French#>;
    } else {
        return nil;
    }

    return configurationUpdate;
}
```

SWIFT

```swift
func managerShouldUpdateLifecycle(toLanguage language:
SDLLanguage) -> SDLLifecycleConfigurationUpdate? {
    let configurationUpdate = SDLLifecycleConfigurationUpdate()

    switch language {
    case .enUs:
        configurationUpdate.appName = <#App Name in English#>
    case .esMx:
        configurationUpdate.appName = <#App Name in Spanish#>
    case .frCa:
        configurationUpdate.appName = <#App Name in French#>
    default:
        return nil
    }

    return configurationUpdate
}
```

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 any 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.

OBJECTIVE-C

```
- (void)hmiLevel:(SDLHMILevel)oldLevel didChangeToLevel:(SDLHMILevel)newLevel {
    if (![newLevel isEqualToEnum:SDLHMILevelNone] && (self.firstHMILevel == SDLHMIFirstStateNone)) {
        // This is our first time in a non-`NONE` state
        self.firstHMILevel = newLevel;
        <#Send static menu RPCs#>
    }

    if ([newLevel isEqualToEnum:SDLHMILevelFull]) {
        <#Send user interface RPCs#>
    } else if ([newLevel isEqualToEnum:SDLHMILevelLimited]) {
        <#Code#>
    } else if ([newLevel isEqualToEnum:SDLHMILevelBackground]) {
        <#Code#>
    } else if ([newLevel isEqualToEnum:SDLHMILevelNone]) {
        <#Code#>
    }
}
```

SWIFT

```
fileprivate var firstHMILevel: SDLHMILevel = .none
func hmiLevel(_ oldLevel: SDLHMILevel, didChangeToLevel newLevel:
SDLHMILevel) {
    if newLevel != .none && firstHMILevel == .none {
        // This is our first time in a non-`NONE` state
        firstHMILevel = newLevel
        <#Send static menu RPCs#>
    }

    switch newLevel {
    case .full:
        <#Send user interface RPCs#>
    case .limited: break
    case .background: break
    case .none: break
    default: break
    }
}
```

Permission Manager

When your app first connects to the head unit, it will receive an `OnPermissionsChange` notification. This notification contains all RPCs the head unit supports and the `hmiLevel` permissions for each RPC. Use the `SDLManager`'s permission manager to check the current permission status of a specific RPC or group of RPCs. If desired, you may also subscribe to get notifications when the RPC(s) permission status changes.

Checking Current Permissions of a Single RPC

OBJECTIVE-C

```
BOOL isAllowed = [self.sdlManager.permissionManager isRPCAllowed:<
#RPC name#>];
```

SWIFT

```
let isAllowed = sdlManager.permissionManager.isRPCAllowed(<#RPC
name#>)
```

Checking Current Permissions of a Group of RPCs

OBJECTIVE-C

```
SDLPermissionGroupStatus groupPermissionStatus = [self.sdlManager.
permissionManager groupStatusOfRPCs:@[<#RPC name#>, <#RPC
name#>]];
NSDictionary *individualPermissionStatuses = [self.sdlManager.
permissionManager statusOfRPCs:@[<#RPC name#>, <#RPC name#
>]];
```

SWIFT

```
let groupPermissionStatus = sdlManager.permissionManager.
groupStatus(ofRPCs:[<#RPC name#>, <#RPC name#>])
let individualPermissionStatuses = sdlManager.permissionManager.
status(ofRPCs:[<#RPC name#>, <#RPC name#>])
```

Observing Permissions

If desired, you can set an observer for a group of permissions. The observer'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, set the `groupType` to `SDLPermissionGroupTypeAllAllowed`.

OBJECTIVE-C

```
SDLPermissionObserverIdentifier observerId = [self.sdlManager.permissionManager addObserverForRPCs:@[<#RPC name#>, <#RPC name#>] groupType:<#SDLPermissionGroupType#> withHandler:^(NSDictionary<SDLPermissionRPCName, NSNumber<SDLBool> *> * _Nonnull change, SDLPermissionGroupStatus status) {
    <#RPC group status changed#>
}];
```

SWIFT

```
let observerId = sdlManager.permissionManager.addObserver(forRPCs: <#RPC name#>, <#RPC name#>, groupType:<#SDLPermissionGroupType#>, withHandler: { (individualStatuses, groupStatus) in
    <#RPC group status changed#>
})
```

Stopping Observation of Permissions

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

OBJECTIVE-C

```
[self.sdlManager.permissionManager removeObserverForIdentifier:
observerId];
```

SWIFT

```
sdlManager.permissionManager.removeObserver(forIdentifier:
observerId)
```

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 occuring 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.

OBJECTIVE-C

```
- (void)audioStreamingState:(nullable SDLAudioStreamingState)
oldState didChangeToState:(SDLAudioStreamingState)newState {
    <#code#>
}
```

SWIFT

```
func audioStreamingState(_ oldState: SDLAudioStreamingState?,
didChangeToState newState: SDLAudioStreamingState) {
    <#code#>
}
```

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.

OBJECTIVE-C

```
- (void)systemContext:(nullable SDLSystemContext)oldContext
didChangeToContext:(SDLSystemContext)newContext {
    <#code#>
}
```

SWIFT

```
func systemContext(_ oldContext: SDLSystemContext?,
didChangeToContext newContext: SDLSystemContext) {
    <#code#>
}
```

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 may 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 Hardware

To connect the example app to Manticore or another emulator, make sure you are on the `TCP Debug` tab and type in the IP address and port, then press "Connect". The button will turn green when you are connected.

To connect the example app to production or debug hardware, make sure you are on the `iAP` tab and press "Connect". The button will turn green when you are connected.

Adaptive Interface Capabilities

Designing for Different Head Units

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. When the app first connects to the SDL Core, a `RegisterAppInterface` RPC will be sent by the SDL Core containing the `displayCapability`, `buttonCapabilites`, etc., properties. You can use this information to determine how to lay out the user interface.

You may access these properties on the `SDLManager.systemCapabilityManager` instance as of SDL iOS library 6.0. More advanced capabilities, such as the `SDLRemoteControlCapability` must be updated through the `systemCapabilityManager`.

System Capability Manager Properties

The `SystemCapabilityManager` is a new feature available as of version 6.0. If using previous versions of the library, you can find most of the `SystemCapabilityManager` properties in the `SDLRegisterAppInterfaceResponse` object. You

will have to manually extract the desired capability from the `SDLManager.registerResponse` property.

PARAMETERS	DESCRIPTION	NOTES
displayCapabilities	Information about the Sync 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.	Check SDLDisplayCapabilities.h for more information
buttonCapabilities	A list of available buttons and whether the buttons support long, short and up-down presses.	Check SDLButtonCapabilities.h for more information
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.	Check SDLSoftButtonCapabilities.h for more information
presetBankCapabilities	If returned, the platform supports custom on-screen presets.	Check SDLPresetBankCapabilities.h for more information
hmiZoneCapabilities	Specifies HMI Zones in the vehicle. There may be a HMI available for back seat passengers as well as front seat passengers.	Check SDLHMIZoneCapabilities.h for more information
speechCapabilities	Contains information about TTS capabilities on the SDL platform. Platforms may support text, SAPI phonemes, LH PLUS phonemes, pre-recorded speech, and silence.	Check SDLSpeechCapabilities.h for more information

PARAMETERS	DESCRIPTION	NOTES
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.	Check SDLPrerecordedSpeech.h for more information
vrCapability	The voice-recognition capabilities of the connected SDL platform. The platform may be able to recognize spoken text in the current language.	Check SDLVRCapabilities.h for more information
audioPassThruCapabilities	Describes the sampling rate, bits per sample, and audio types available.	Check SDLAudioPassThruCapabilities.h for more information
hmiCapabilities	Returns whether or not the app can support built-in navigation and phone calls.	Check SDLHMICapabilities.h for more information
navigationCapability	Describes the built-in vehicle navigation system's APIs	Check SDLNavigationCapability.h for more information
phoneCapability	Describes the built-in phone calling capabilities of the IVI system.	Check SDLPhoneCapability.h for more information
videoStreamingCapability	Describes the abilities of the head unit to video stream projection applications	Check SDLVideoStreamingCapability.h for more information
remoteControlCapability	Describes the abilities of an app to control built-in aspects of the IVI system	Check SDLRemoteControlCapability.h for more information

The Register App Interface RPC

The `RegisterAppInterface` response contains information about the display type, the type of images supported, the number of text fields supported, the HMI display language, and a lot of other useful properties. The table below has a list of properties beyond those available on the `SystemCapabilityManager` returned by the `RegisterAppInterface` response. Each property is optional, so you may not get information for all the parameters in the table.

PARAMETERS	DESCRIPTION	NOTES
syncMsgVersion	Specifies the version number of the SDL V4 interface. This is used by both the application and SDL to declare what interface version each is using.	Check SDLSyncMsgVersion.h for more information
language	The currently active voice-recognition and text-to-speech language on Sync.	Check SDLLanguage.h for more information
vehicleType	The make, model, year, and the trim of the vehicle.	Check SDLVehicleType.h for more information
supportedDiagModes	Specifies the white-list of supported diagnostic modes (0x00-0xFF) capable for DiagnosticMessage requests. If a mode outside this list is requested, it will be rejected.	Check SDLDiagnosticMessage.h for more information
sdlVersion	The SmartDeviceLink version	String
systemSoftwareVersion	The software version of the system that implements the SmartDeviceLink core	String

Image Specifics

Image File Type

Images may be formatted as PNG, JPEG, or BMP. Check the `RegisterAppInterfaceResponse.displayCapability` properties to find out what image formats the head unit supports.

Image Sizes

If an image is uploaded that is larger than the supported size, that image will be scaled down to accommodate. All image sizes are available from the `SDLManager`'s `registerResponse` property once in the completion handler for `startWithReadyHandler`.

IMAGE NAME	USED IN RPC	DETAILS	HEIGHT	WIDTH	TYPE
softButtonImage	Show	Will be shown on softbuttons on the base screen	70px	70px	png, jpg, bmp
choiceImage	CreateInteractionChoiceSet	Will be shown in the manual part of an performInteraction either big (ICON_ONLY) or small (LIST_ONLY)	70px	70px	png, jpg, bmp
choiceSecondaryImage	CreateInteractionChoiceSet	Will be shown on the right side of an entry in (LIST_ONLY) performInteraction	35px	35px	png, jpg, bmp
vrHelpItem	SetGlobalProperties	Will be shown during voice interaction	35px	35px	png, jpg, bmp
menuIcon	SetGlobalProperties	Will be shown on the “More...” button	35px	35px	png, jpg, bmp

IMAGE NAME	USED IN RPC	DETAILS	HEIGHT	WIDTH	TYPE
cmdIcon	AddCommand	Will be shown for commands in the "More..." menu	35px	35px	png, jpg, bmp
appIcon	SetAppIcon	Will be shown as Icon in the "Mobile Apps" menu	70px	70px	png, jpg, bmp
graphic	Show	Will be shown on the basescreen as cover art	185px	185px	png, jpg, bmp

Main Screen Templates

Each car manufacturer supports a set of templates for the user interface. These templates determine the position and size of the text, images, and buttons on the screen. A list of supported templates is sent in `SDLManager.systemCapabilityManager.displayCapabilities.templatesAvailable`.

To change a template at any time, send a `SDLSetDisplayLayout` RPC to the SDL Core. If you want to ensure that the new template is used, wait for a response from the SDL Core before sending any more user interface RPCs.

OBJECTIVE-C

```
SDLSetDisplayLayout* display = [[SDLSetDisplayLayout alloc]
initWithPredefinedLayout:SDLPredefinedLayoutGraphicWithText];
[self.sdlManager sendRequest:display withResponseHandler:^(
SDLRPCRequest *request, SDLRPCResponse *response, NSError *error)
{
    if ([response.resultCode isEqualToEnum:SDLResultSuccess]) {
      // The template has been set successfully
    }
}];
```

SWIFT

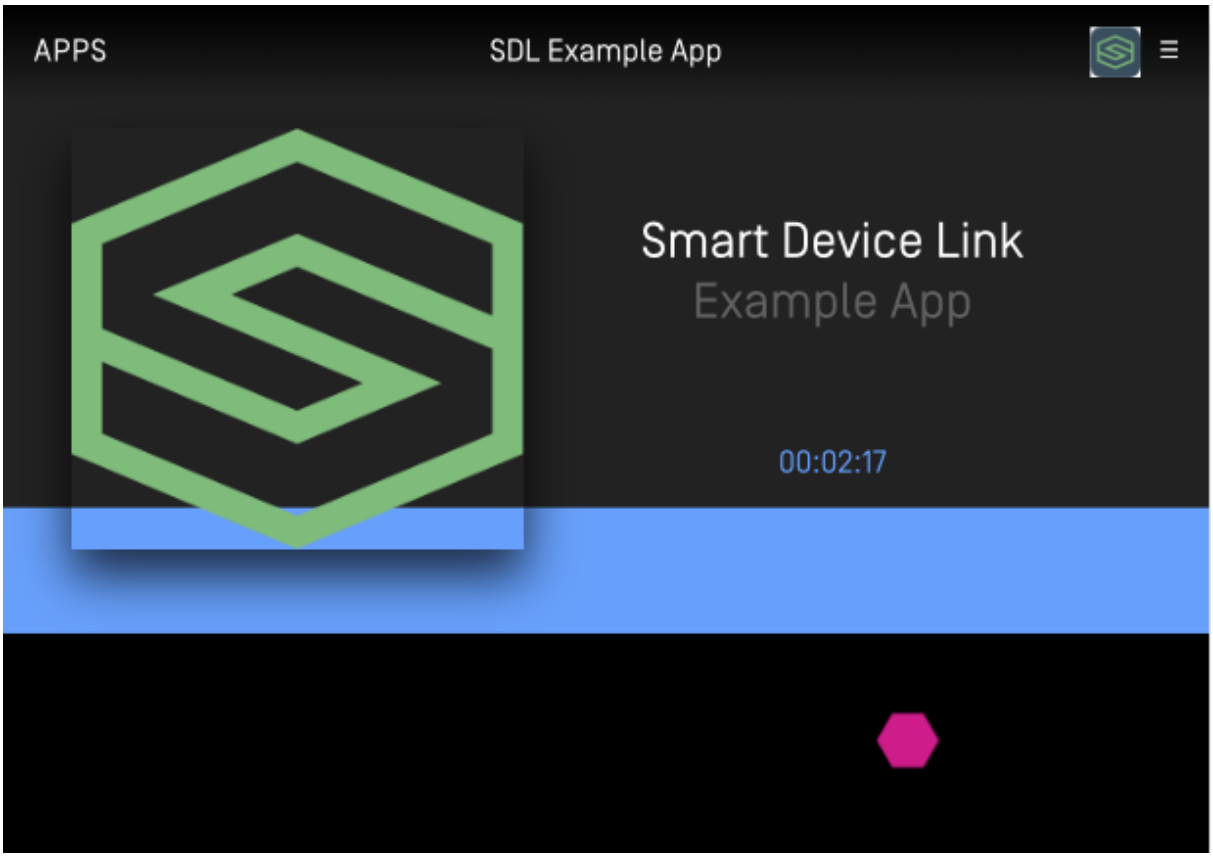
```
let display = SDLSetDisplayLayout(predefinedLayout: .graphicWithText)
sdlManager.send(request: display) { (request, response, error) in
    if response?.resultCode == .success {
        // The template has been set successfully
    }
}
```

Available Templates

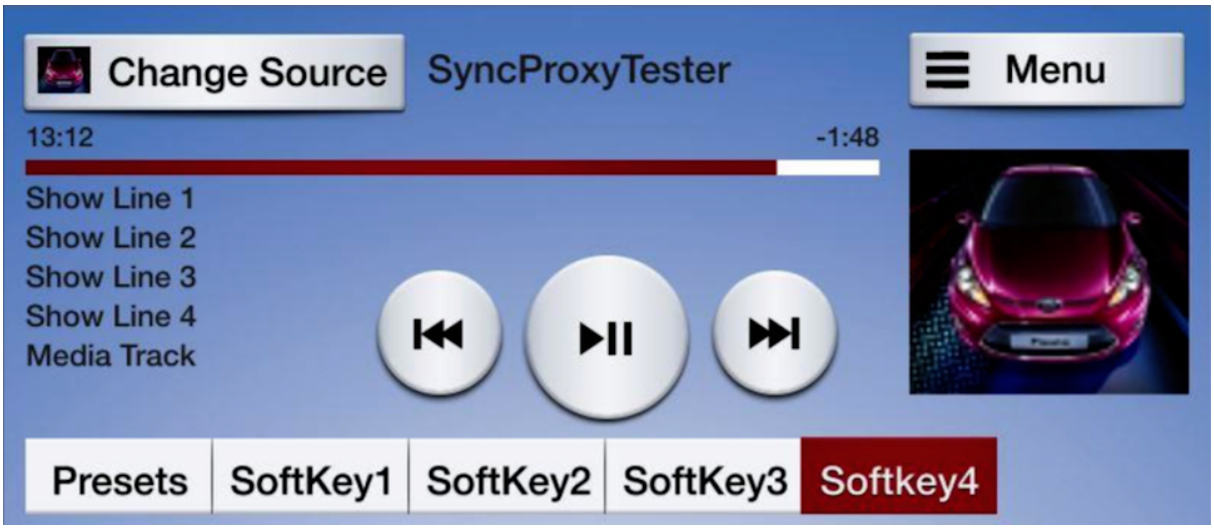
There are fifteen standard templates to choose from, however some head units may only support a subset of these templates. Please check `SystemCapability Manager` for the supported templates. The following examples show how templates will appear on the Generic HMI and Ford's SYNC 3 HMI.

MEDIA - WITH AND WITHOUT PROGRESS BAR

Generic HMI



Ford HMI



NON-MEDIA - WITH AND WITHOUT SOFT BUTTONS

Generic HMI

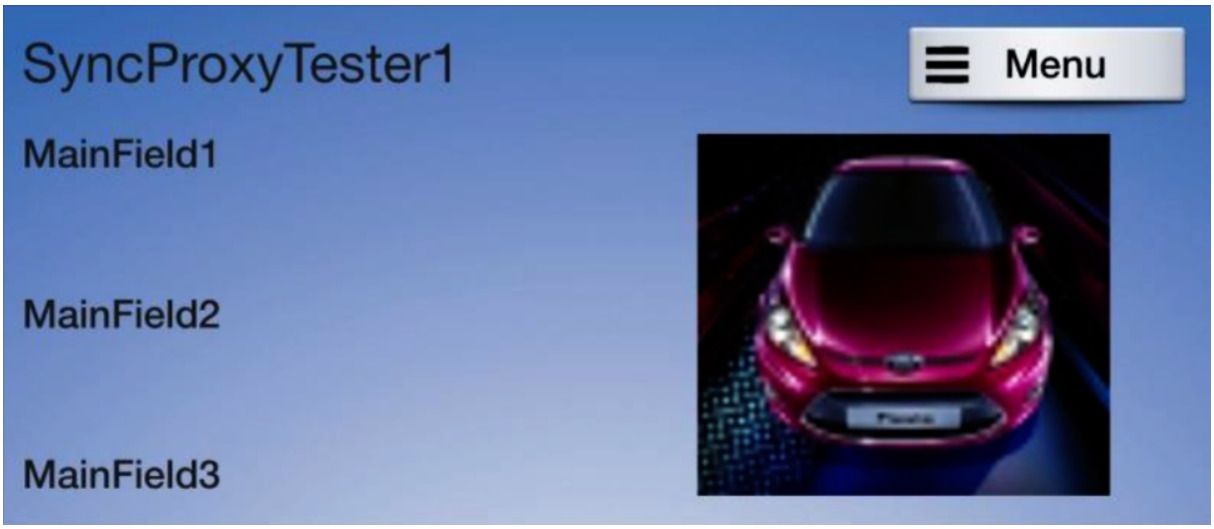
Ford HMI

GRAPHIC_WITH_TEXT

Ford HMI

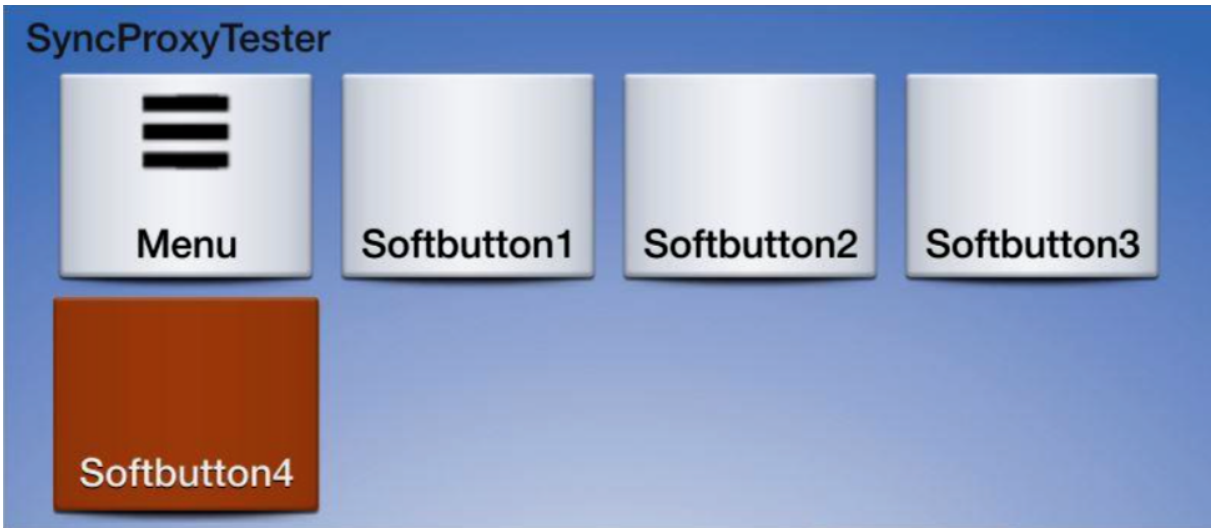
TEXT_WITH_GRAPHIC

Ford HMI



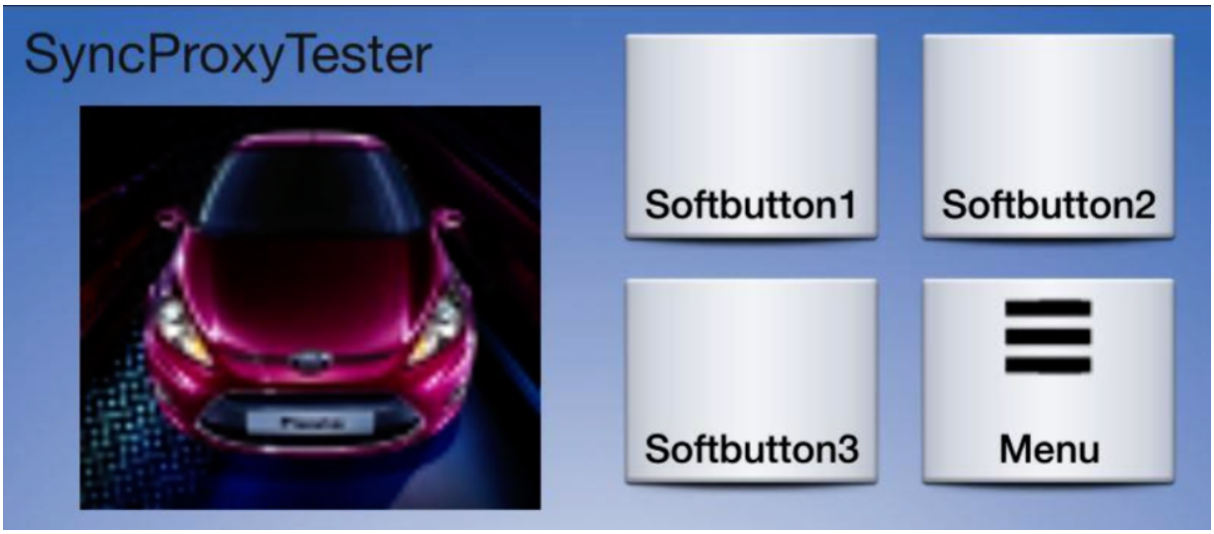
TILES_ONLY

Ford HMI



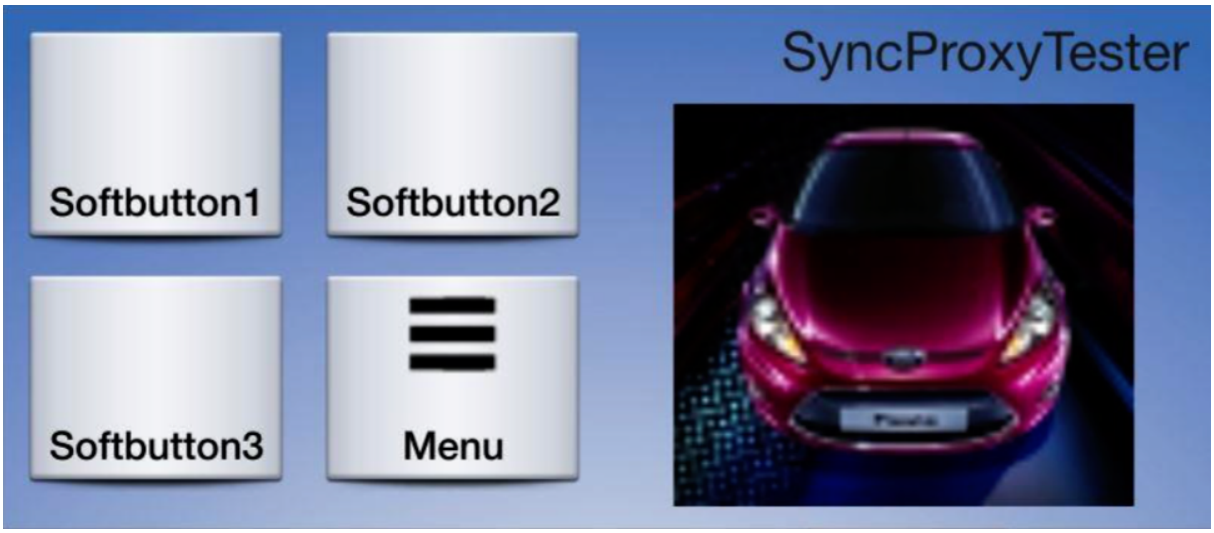
GRAPHIC_WITH_TILES

Ford HMI



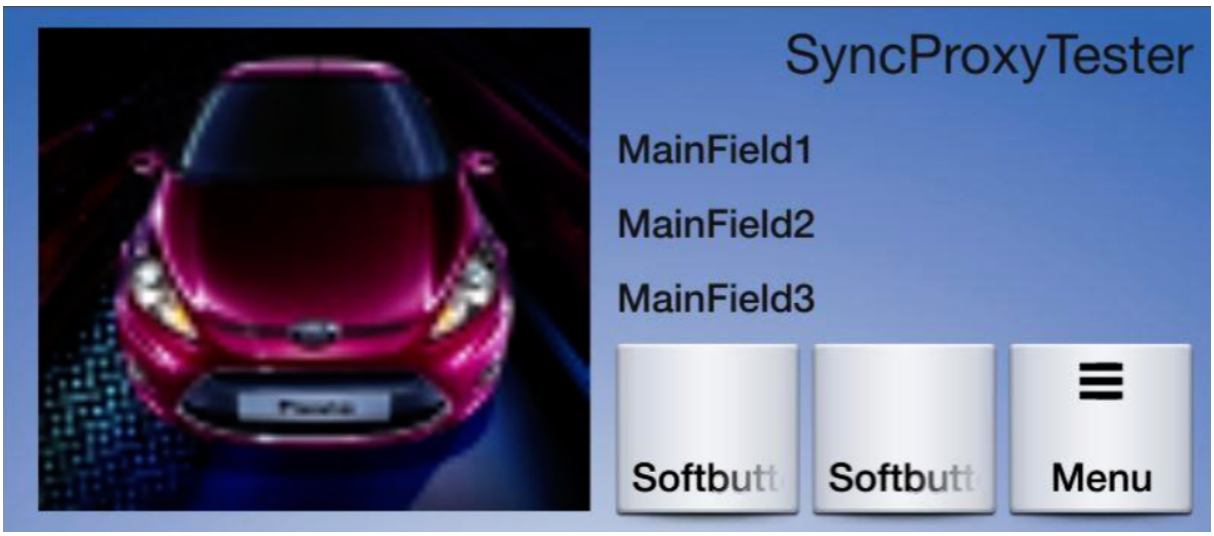
TILES_WITH_GRAPHIC

Ford HMI



GRAPHIC_WITH_TEXT_AND_SOFTBUTTONS

Ford HMI



TEXT_AND_SOFTBUTTONS_WITH_GRAPHIC

Ford HMI

GRAPHIC_WITH_TEXTBUTTONS

Ford HMI

DOUBLE_GRAPHIC_SOFTBUTTONS

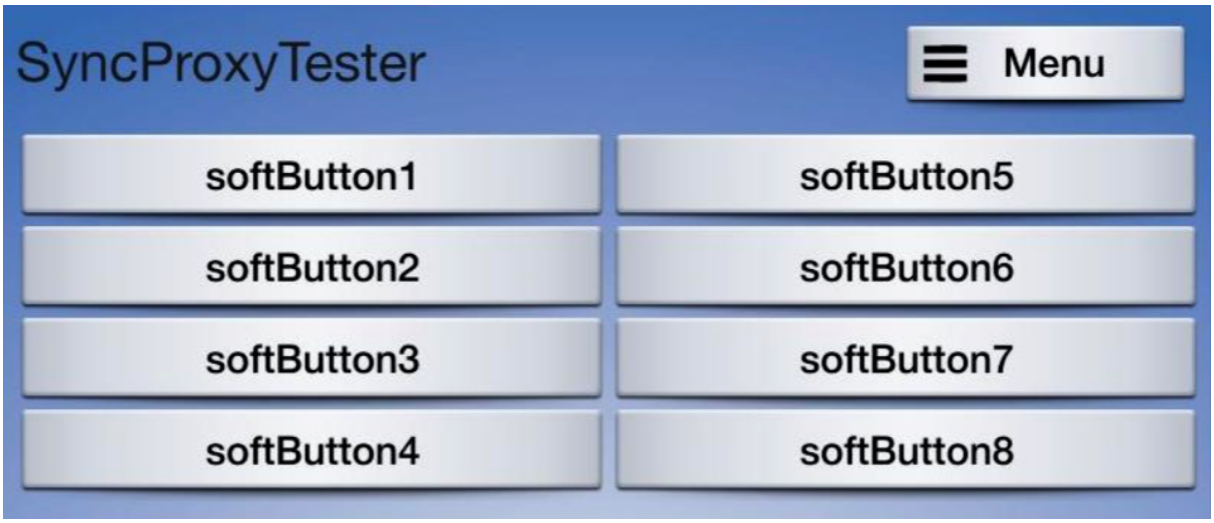
Ford HMI

TEXTBUTTONS_WITH_GRAPHIC

Ford HMI

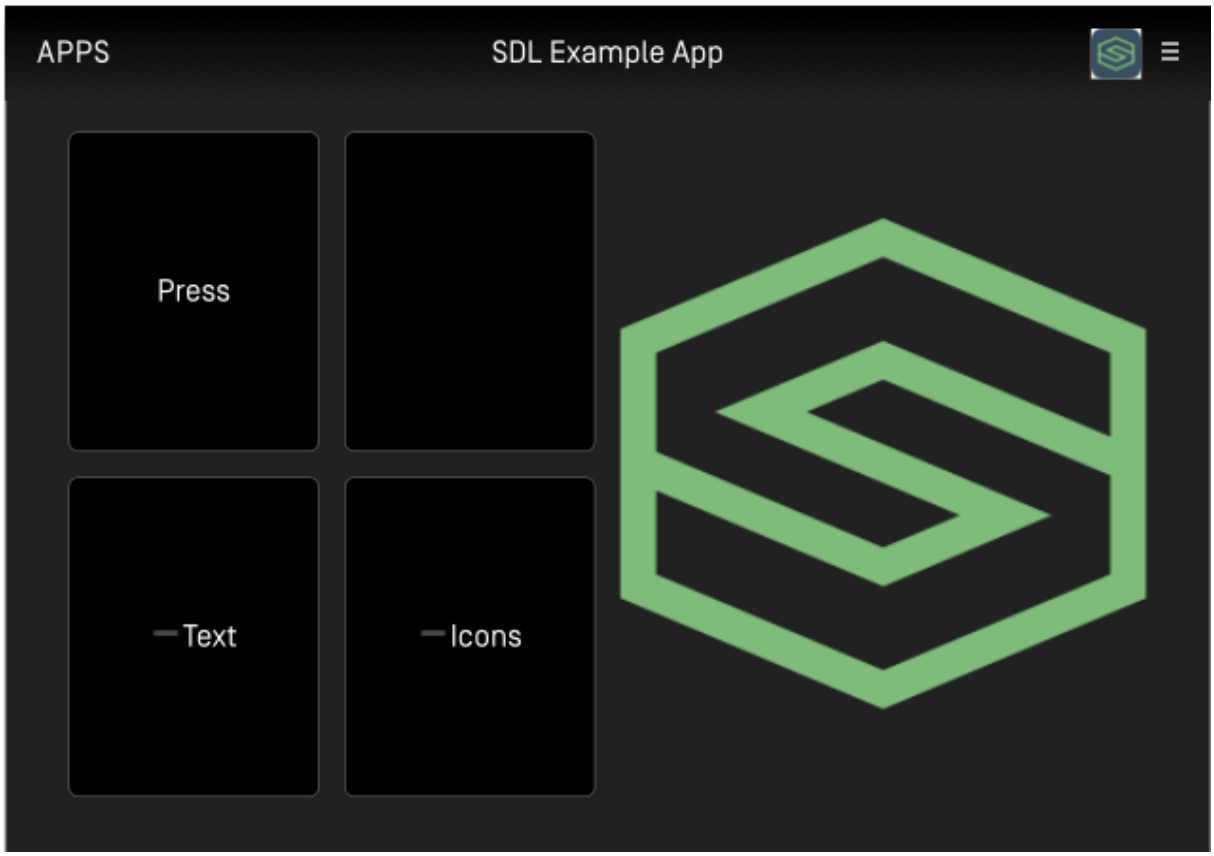
TEXTBUTTONS_ONLY

Ford HMI



LARGE_GRAPHIC_WITH_SOFTBUTTONS

Generic HMI



Ford HMI

LARGE_GRAPHIC_ONLY

Generic HMI

Text, Images, and Buttons

Template Fields

The `SDLScreenManager` is a manager for easily creating and sending text, images and soft buttons for your SDL app. To update the UI, simply give the manager the new UI data and sandwich the update between the manager's `beginUpdates` and `endUpdatesWithCompletionHandler:` methods.

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
primaryGraphic	The primary image in a template that supports images
secondaryGraphic	The second image in a template that supports multiple images
textAlignment	The text justification for the text fields. The text alignment can be left, center, or right
softButtonObjects	An array of buttons. Each template supports a different number of soft buttons
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`

OBJECTIVE-C

```
[self.sdlManager.screenManager beginUpdates];

self.sdlManager.screenManager.textField1 = @"<#Line 1 of Text#>";
self.sdlManager.screenManager.textField2 = @"<#Line 2 of Text#>";
self.sdlManager.screenManager.primaryGraphic = [SDLArtwork
persistentArtworkWithImage:[UIImage imageNamed:@"<#Image
Name#>"] asImageFormat:<#SDLArtworkImageFormat#>]
SDLSoftButtonObject *softButton = [[SDLSoftButtonObject alloc]
initWithName:@"<#Soft Button Name#>" state:[[SDLSoftButtonState
alloc] initWithStateName:@"<#Soft Button State Name#>" text:
@"<#Button Text#>" artwork:<#SDLArtwork#>] handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_Nullable buttonEvent) {
    if (buttonPress == nil) { return; }
    <#Button Selected#>
}];
self.sdlManager.screenManager.softButtonObjects = @[softButton];

[self.sdlManager.screenManager endUpdatesWithCompletionHandler:^(
NSError * _Nullable error) {
    if (error != nil) {
        <#Error Updating UI#>
    } else {
        <#Update to UI was Successful#>
    }
}];
```

SWIFT

```
sdlManager.screenManager.beginUpdates()

sdlManager.screenManager.textField1 = "<#Line 1 of Text#>"
sdlManager.screenManager.textField2 = "<#Line 2 of Text#>"
sdlManager.screenManager.primaryGraphic = <#SDLArtwork#>
sdlManager.screenManager.softButtonObjects = [<#SDLButtonObject#
>, <#SDLButtonObject#>]

sdlManager.screenManager.endUpdates { (error) in
    if error != nil {
        <#Error Updating UI#>
    } else {
        <#Update to UI was Successful#>
    }
}
```

Soft Button Objects

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 upload all the states on initialization.

Updating the Soft Button State

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

OBJECTIVE-C

```
SDLSoftButtonState *softButtonState1 = [[SDLSoftButtonState alloc]
initWithStateName:@"<#Soft Button State Name#>" text:
@"<#Button Label Text#>" artwork:<#SDLArtwork#>];
SDLSoftButtonState *softButtonState2 = [[SDLSoftButtonState alloc]
initWithStateName:@"<#Soft Button State Name#>" text:
@"<#Button Label Text#>" artwork:<#SDLArtwork#>];
SDLSoftButtonObject *softButtonObject = [[SDLSoftButtonObject alloc]
initWithName:@"<#Soft Button Object Name#>" states:@[
softButtonState1, softButtonState2] initialStateName:@"<#Soft Button
State Name#>" handler:^(SDLOnButtonPress * _Nullable buttonPress,
SDLOnButtonEvent * _Nullable buttonEvent) {
    if (buttonPress == nil) { return; }
    <#Button Selected#>
}];
self.sdlManager.screenManager.softButtonObjects = @[
softButtonObject];

// Transition to a new state
SDLSoftButtonObject *retrievedSoftButtonObject = [self.sdlManager.
screenManager softButtonObjectNamed:@"<#Soft Button Object
Name#>"];
[retrievedSoftButtonObject transitionToNextState];
```

SWIFT

```
let softButtonState1 = SDLSoftButtonState(stateName: "<#Soft Button
State Name#>", text: "<#Button Label Text#>", artwork: <#
SDLArtwork#>)
let softButtonState2 = SDLSoftButtonState(stateName: "<#Soft Button
State Name#>", text: "<#Button Label Text#>", artwork: <#
SDLArtwork#>)
let softButtonObject = SDLSoftButtonObject(name: "<#Soft Button
Object Name#>", states: [softButtonState1, softButtonState2],
initialStateName: "") { (buttonPress, buttonEvent) in
    guard buttonPress != nil else { return }
    <#Button Selected#>
}
sdlManager.screenManager.softButtonObjects = [softButtonObject]

// Transition to a new state
let retrievedSoftButtonObject = sdlManager.screenManager.
softButtonObjectNamed("<#Soft Button Object Name#>")
retrievedSoftButtonObject?.transitionToNextState()
```

Deleting Soft Buttons

To delete soft buttons, simply pass the `SDLScreenManager` an empty array of soft buttons.

Template Images

As of SDL iOS library v6.1, when connected to a remote system running SDL Core 5.0+, you may be able to use template images. 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.

OBJECTIVE-C

```
UIImage *image = [[UIImage imageNamed:<#String#>]
imageWithRenderingMode:UIImageRenderingModeAlwaysTemplate];
SDLArtwork *artwork = [SDLArtwork artworkWithImage:image
asImageFormat:SDLArtworkImageFormatPNG];
```

SWIFT

```
let image = UIImage(named: <#T##String#>)?.withRenderingMode(.
alwaysTemplate)
let artwork = SDLArtwork(image: image, persistent: true, as: .PNG)
```

Static Icons

Static icons are pre-existing images on the remote system that you may reference and use in your own application. Static icons will be supported by the screen manager in a future update. Until then, you must send them using RPC request APIs `Image` and `Show`.

OBJECTIVE-C

```
SDLImage *image = [[SDLImage alloc] initWithStaticIconName:<#(
nonnull SDLStaticIconName)#>];
SDLShow *show = [[SDLShow alloc] init];
show.graphic = image;
[self.sdlManager sendRequest:show];
```

SWIFT

```
let image = SDLImage(staticIconName: <#SDLStaticIconName#>)
let show = SDLShow()
show.graphic = image
sdlManager.send(show)
```

Using RPCs

If you don't want to use the screen manager, you can use raw RPC requests using the `Show` RPC.

Subscribing to System Buttons

Subscribe buttons are used to detect changes to hard buttons located in the car's center console or steering wheel. You can subscribe to the following hard buttons:

BUTTON	TEMPLATE	BUTTON TYPE
Play / Pause	media template only	soft button and hard button
Ok	any template	soft button and hard button
Seek left	media template only	soft button and hard button
Seek right	media template only	soft button and hard button
Tune up	media template only	hard button
Tune down	media template only	hard button
Preset 0-9	any template	hard button
Search	any template	hard button

NOTE

There is no way to customize a subscribe button's image or text.

Audio-Related Buttons

The play/pause, seek left, seek right, tune up, and tune down subscribe buttons can only be used in the `MEDIA` template. Depending on the manufacturer of the head unit, the subscribe button might also show up as a soft button in the media template. For example, the SYNC 3 HMI will add the ok, seek right, and seek left soft buttons to the media template when you subscribe to those buttons. You will automatically be assigned the media template if you set your app's configuration `appType` to `MEDIA`.

NOTE

Before iOS library v6.1 and SDL Core 5.0, `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 do the right thing based on the version of Core to which you are subscribed.

OBJECTIVE-C

```
SDLSubscribeButton *subscribeButton = [[SDLSubscribeButton alloc]
initWithButtonName:SDLButtonNamePlayPause handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_Nullable buttonEvent) {
    <#subscribe button selected#>
}];
[manager sendRequest:subscribeButton withResponseHandler:^(
__kindof SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse
 * _Nullable response, NSError * _Nullable error) {
    if (error != nil) { return; }
    <#subscribe button sent successfully#>
}];
```

SWIFT

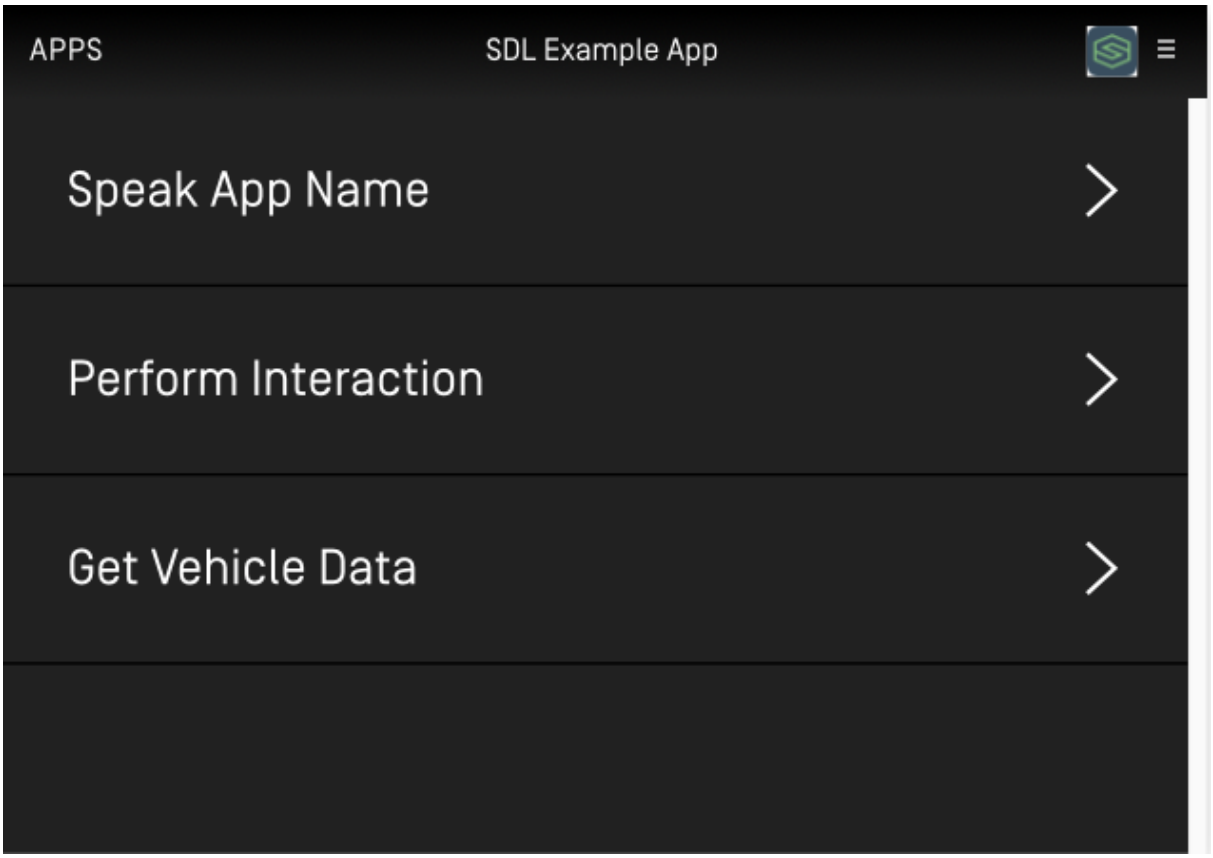
```
let subscribeButton = SDLSubscribeButton(buttonName: .ok) { (
buttonPress, buttonEvent) in
    <#subscribe button selected#>
}
sdlManager.send(request: subscribeButton) { (request, response, error)
 in
    guard error == nil else { return }
    <#subscribe button sent successfully#>
}
```

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 and Keyboards section.

Every template has a main menu button. The position of this button varies between templates, and can not be removed from the template. The default menu is initially empty except for an "Exit {App Name}" button. Items can be added to the menu at the root level or to a submenu.

Generic HMI



Ford HMI

Adding Menu Items

As of iOS library v6.0, the best way to create and update your menu is 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.

OBJECTIVE-C

```
// Create the menu cell
SDLMenuCell *cell = [[SDLMenuCell alloc] initWithTitle:<#Menu Item Text#> icon:<#Menu Item Artwork#> voiceCommands:@[<#Menu Item #>] handler:^(SDLTriggerSource  _Nonnull triggerSource) {
    // Menu item was selected, check the `triggerSource` to know if the user used touch or voice to activate it
    <#Handle the cell's selection#>
}];

self.sdlManager.screenManager.menu = @[cell];
```

SWIFT

```
// Create the menu cell
let cell = SDLMenuCell(title: <#String#>, icon: <#SDLArtwork?#>,
voiceCommands: <#[String]?#>) { (triggerSource: SDLTriggerSource)
in
    // Menu item was selected, check the `triggerSource` to know if the
user used touch or voice to activate it
    <#Handle the Cell's Selection#>
}

self.sdlManager.screenManager.menu = [cell]
```

Adding Submenus

Adding a submenu is as simple as adding subcells to an `SDLMenuCell`. This can currently only be done with one layer of subcells, and automatically displays the submenu when selected.

OBJECTIVE-C

```
// Create the inner menu cell
SDLMenuCell *cell = [[SDLMenuCell alloc] initWithTitle:<#Menu Item
Text#> icon:<#Menu Item Artwork#> voiceCommands:@[<#Menu
Item #>] handler:^(SDLTriggerSource  _Nonnull triggerSource) {
    // Menu item was selected, check the `triggerSource` to know if the
user used touch or voice to activate it
    <#Handle the cell's selection#>
}];

// Create and set the submenu cell
SDLMenuCell *submenuCell = [[SDLMenuCell alloc] initWithTitle:<#
Menu Item Text#> icon:<#SDLArtwork#> subCells:@[cell]];
self.sdlManager.screenManager.menu = @[submenuCell];
```

SWIFT

```
// Create the inner menu cell
let cell = SDLMenuCell(title: <#T##String#>, icon: <#T##SDLArtwork
?#>, voiceCommands: <#T##[String]?#>) { (triggerSource:
SDLTriggerSource) in
    // Menu item was selected, check the `triggerSource` to know if the
user used touch or voice to activate it
    <#code#>
}

let submenuCell = SDLMenuCell(title: <#T##String#>, icon: <#
SDLArtwork#>, subCells:<#T##[SDLMenuCell]#>)
self.sdlManager.screenManager.menu = @[submenuCell]
```

Artworks

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 Menu Items

When using the screen manager, this will be intelligently handled 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. The screen manager will handle deleting and adding new commands and submenus for you. If you are

doing this manually, you must use the `SDLDeleteCommand` and `SDLDeleteSubMenu` RPCs, passing the `cmdID`s you wish to delete.

Using RPCs

The `SDLAddCommand` RPC can be used to add items to the root menu or to a submenu. Each `SDLAddCommand` 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 `SDLAddSubMenu` 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 `SDLAddCommand` RPC for each item in the submenu.

Voice Commands

Voice commands are global commands available anywhere on the head unit to users of your app. If your app has left the HMI state of `NONE` because the user has interacted with your app, they may speak the commands you have setup and trigger actions in your app. How these commands are triggered (and whether they are supported at all) will depend on the head unit you connect to, but you don't have to worry about those intracacies when setting up your global commands.

You have the ability to create voice command shortcuts to your Main Menu cells, and it is recommended that you do so. If you have additional functions that you wish to make available as voice commands that are not available as menu cells, you can create pure global voice commands.

> NOTE
>
> We recommend creating global voice commands for common actions such as the actions performed by your Soft Buttons.

You simply must create and set `SDLVoiceCommand` objects to the `voiceCommands` array on the screen manager.

OBJECTIVE-C

```
// Create the voice command
SDLVoiceCommand *voiceCommand = [[SDLVoiceCommand alloc] initWithVoiceCommands:<#(nonnull NSArray<NSString *> *)#> handler:<#^(void)handler#>];

self.sdlManager.screenManager.voiceCommands = @[voiceCommand];
```

SWIFT

```
// Create the voice command
let voiceCommand = SDLVoiceCommand(voiceCommands: <#T##[String]#>) {
    <#code#>
}

self.sdlManager.screenManager.voiceCommands = [voiceCommand]
```

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.

Popup Menus and Keyboards

SDL supports modal menus and keyboards. These are requests for input from the user based on a list of options you present to the user – that they can respond to via touch or voice (if supported) – or via their own keyboard input.

There are several advantages and disadvantages to this kind of menu compared to the main menu. The main menu should remain more static and should not be updated often and only in predictable ways. The main menu is the best way to perform navigation for your app. By contrast, a popup menu is better for a selection of options for your app, and allows for a keyboard to be available for search or other user input.

Presenting a Popup Menu

You may think of presenting a popup menu as presenting a modal `UITableViewController` to request input from the user. You may chain together menus to drill down, however, it is recommended to do so judiciously, as requesting too much input from the driver while he is driving will be 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
Present Keyboard	A keyboard shows up immediately in the HMI

Creating Cells

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

> 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.

OBJECTIVE-C

```
SDLChoiceCell *cell = [[SDLChoiceCell alloc] initWithText:<#(nonnull
NSString *)#>];
SDLChoiceCell *fullCell = [[SDLChoiceCell alloc] initWithText:<#(
nonnull NSString *)#> secondaryText:<#(nullable NSString *)#>
tertiaryText:<#(nullable NSString *)#> voiceCommands:<#(nullable
NSArray<NSString *> *)#> artwork:<#(nullable SDLArtwork *)#>
secondaryArtwork:<#(nullable SDLArtwork *)#>];
```

SWIFT

```
let cell = SDLChoiceCell(text: <#T##String#>)
let cell = SDLChoiceCell(text: <#T##String#>, secondaryText: <#T##
String?#>, tertiaryText: <#T##String?#>, voiceCommands: <#T##[
String]?#>, artwork: <#T##SDLArtwork?#>, secondaryArtwork: <#T
##SDLArtwork?#>)
```

Preloading Cells

If you know what some or all cells should contain before they are used, you can "preload" these cells in order to speed up their presentation at a later time. The cells you preload may be used individually or a group.

OBJECTIVE-C

```
[self.sdlManager.screenManager preloadChoices:<#(nonnull NSArray<
SDLChoiceCell *> *)#> withCompletionHandler:^(NSError * _Nullable
error) {
    <#code#>
}];
```

SWIFT

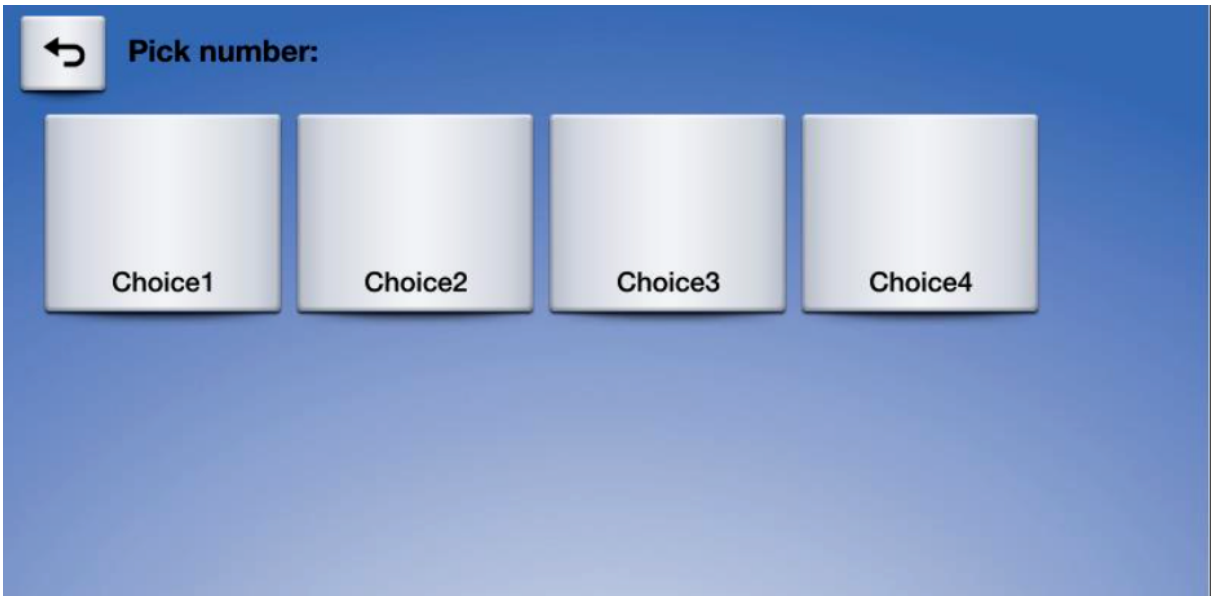
```
sdlManager.screenManager.preloadChoices(<#T##choices: [
SDLChoiceCell]##[SDLChoiceCell]#>) { (error) in
   <#code#>
}
```

Presenting a Menu

Whether or not you preloaded cells, you may present a menu. If you did not preload cells, calling a `present` API will cause them to be preloaded and then presented once they are available. Therefore, this call may take longer than if the cells were preloaded earlier in the app's lifecycle. On later presentations using the same cells, it will reuse those cells (unless you deleted them of course), so later presentations will be faster.

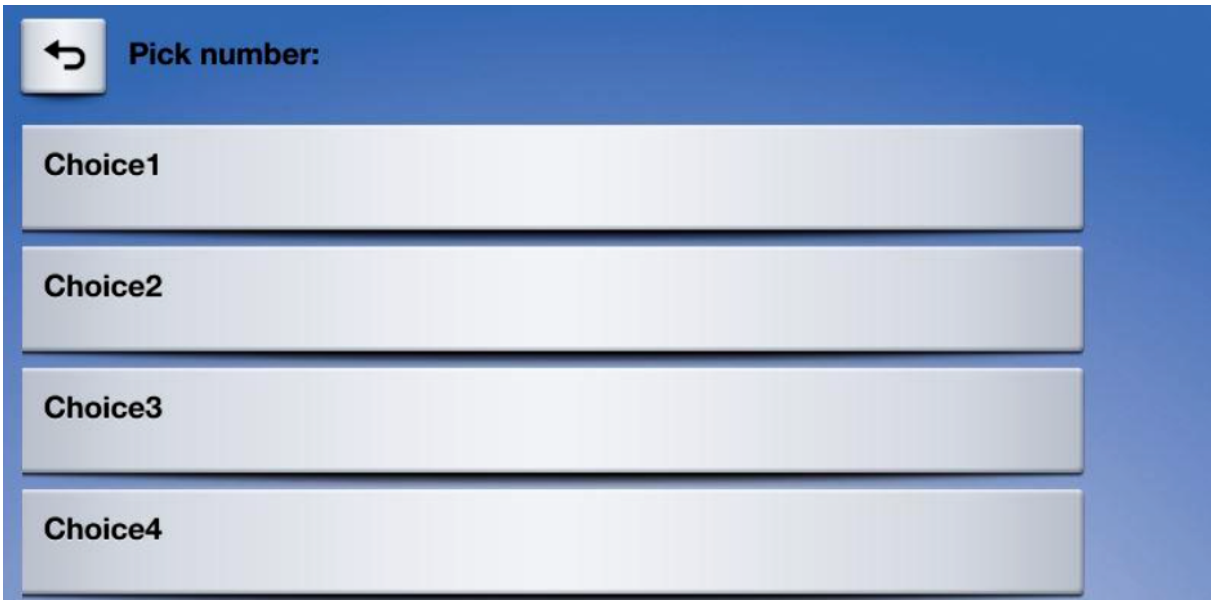
MENU - ICON

Ford HMI



MENU - LIST

Ford HMI



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.

OBJECTIVE-C

```
SDLChoiceSet *choiceSet = [[SDLChoiceSet alloc] initWithTitle:<#(
nonnull NSString *)#> delegate:<#(nonnull id<SDLChoiceSetDelegate
>)#> layout:<#(SDLChoiceSetLayout)#> timeout:<#(NSTimeInterval)
#> initialPromptString:<#(nullable NSString *)#> timeoutPromptString
:<#(nullable NSString *)#> helpPromptString:<#(nullable NSString *)#
> vrHelpList:<#(nullable NSArray<SDLVRHelpItem *> *)#> choices:<#
(nonnull NSArray<SDLChoiceCell *> *)#>];
```

SWIFT

```
let choiceSet = SDLChoiceSet(title: <#T##String#>, delegate: <#T##
SDLChoiceSetDelegate#>, layout: <#T##SDLChoiceSetLayout#>,
timeout: <#T##TimeInterval#>, initialPromptString: <#T##String?#>
, timeoutPromptString: <#T##String?#>, helpPromptString: <#T##
String?#>, vrHelpList: <#T##[SDLVRHelpItem]?#>, choices: <#T##[
SDLChoiceCell]#>)
```

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.

OBJECTIVE-C

```
#pragma mark - SDLChoiceSetDelegate

- (void)choiceSet:(SDLChoiceSet *)choiceSet didSelectChoice:(
SDLChoiceCell *)choice withSource:(SDLTriggerSource)source
atRowIndex:(NSUInteger)rowIndex {
    <#Code#>
}

- (void)choiceSet:(SDLChoiceSet *)choiceSet didReceiveError:(NSError *)
error {
    <#Code#>
}
```

SWIFT

```
extension <#Class Name#>: SDLChoiceSetDelegate {
    func choiceSet(_ choiceSet: SDLChoiceSet, didSelectChoice choice:
SDLChoiceCell, withSource source: SDLTriggerSource, atRowIndex
rowIndex: UInt) {
        <#Code#>
    }

    func choiceSet(_ choiceSet: SDLChoiceSet, didReceiveError error:
Error) {
        <#Code#>
    }
}
```

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 `SDLChoiceCell` you should choose `SDLInteractionModeManualOnly`. If `vrCommands` are available, you may choose `SDLInteractionModeVoiceRecognitionOnly` or `SDLInteractionModeBoth`.

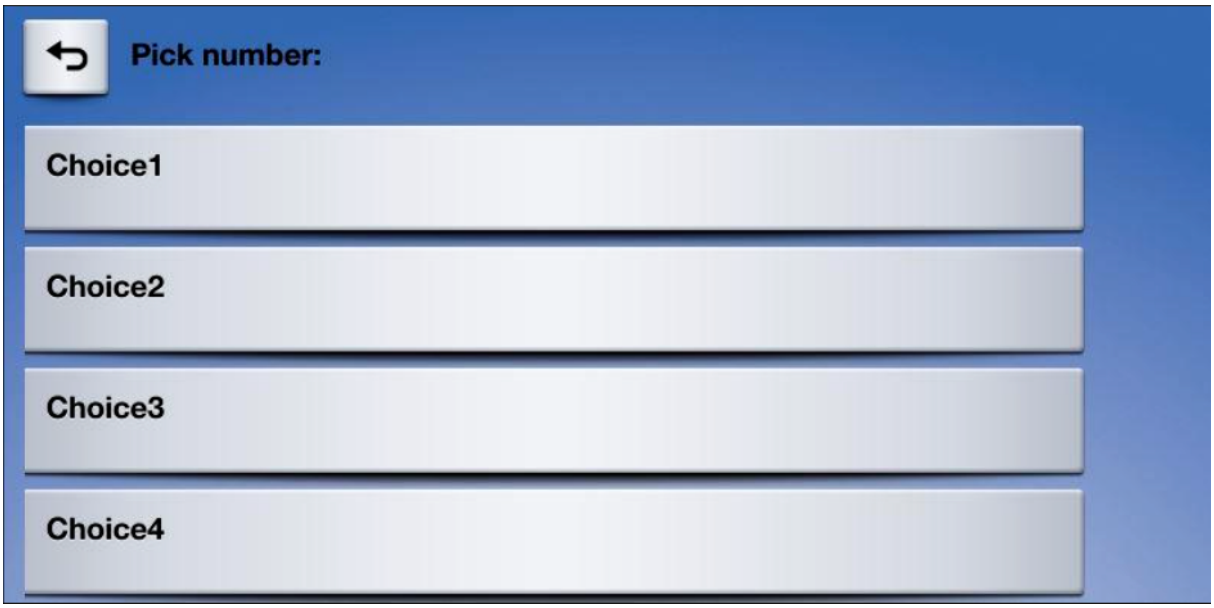
You may want to choose this based on the trigger source leading to the menu being presented. For example, if the menu was presented via the user touching the screen, you may want to use a `mode` of `.manualOnly` or `.both`, but if the menu was presented via the user speaking a voice command, you may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

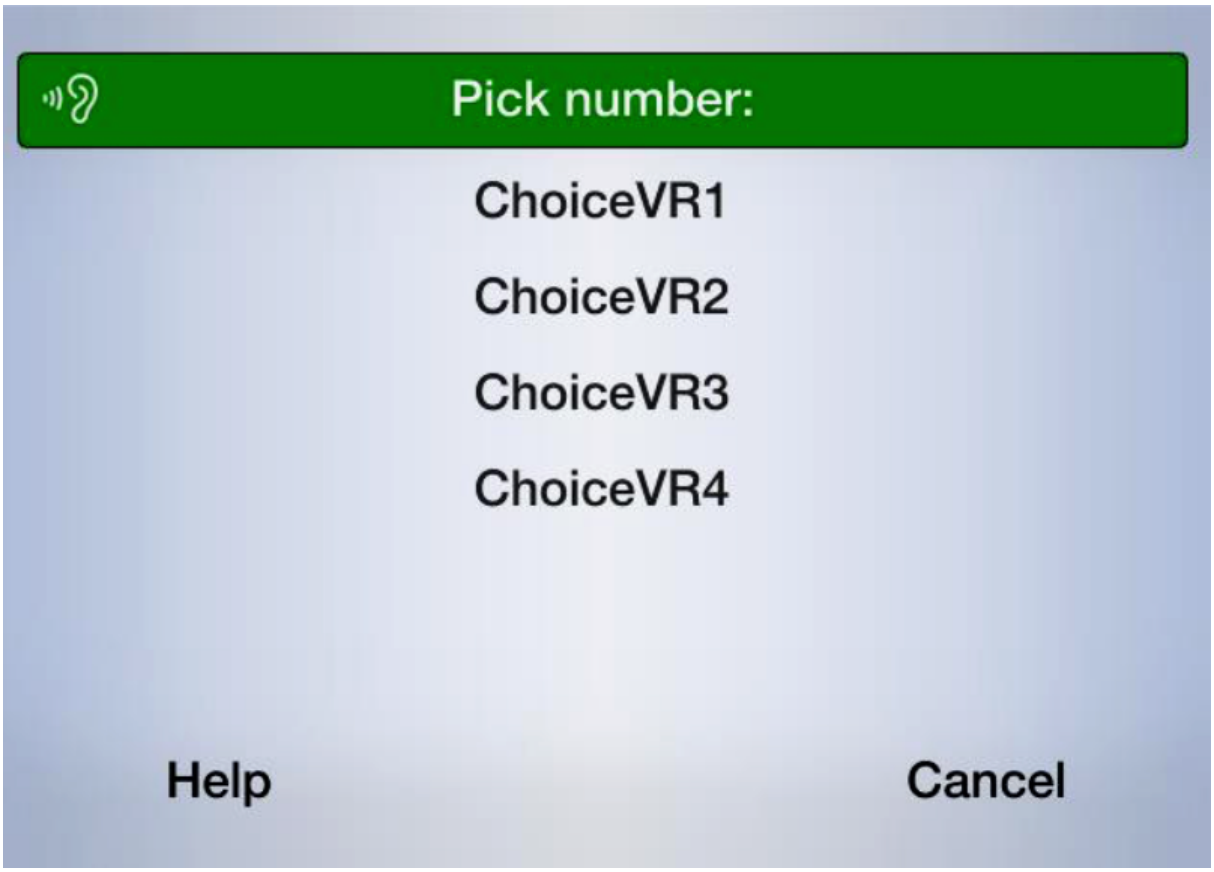
MENU - MANUAL ONLY

Ford HMI



MENU - VOICE ONLY

Ford HMI



OBJECTIVE-C

```
[self.manager.screenManager presentChoiceSet:<#(nonnull
SDLChoiceSet *)#> mode:<#(nonnull SDLInteractionMode)#>];
```

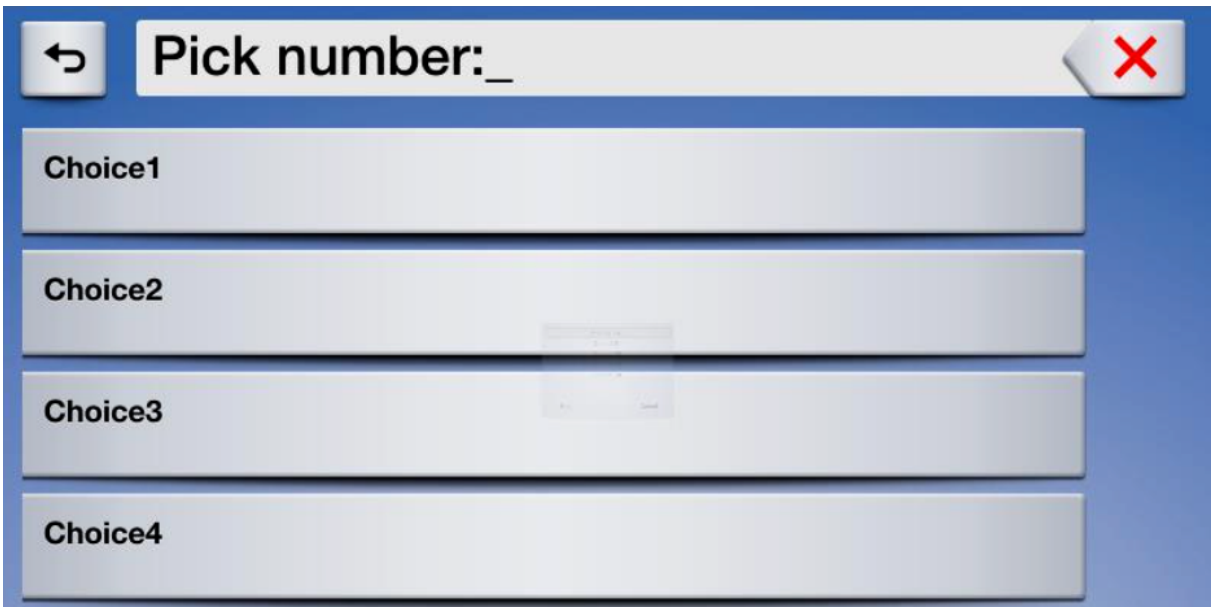
SWIFT

```
manager.screenManager.present(<#T##choiceSet: SDLChoiceSet##
SDLChoiceSet#>, mode: <#T##SDLInteractionMode#>)
```

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 portion of this menu, see *Presenting a Keyboard* below.

Ford HMI



OBJECTIVE-C

```
[self.sdlManager.screenManager presentSearchableChoiceSet:<#(
nonnull SDLChoiceSet *)#> mode:<#(nonnull SDLInteractionMode)#>
withKeyboardDelegate:<#(nonnull id<SDLKeyboardDelegate>)#>];
```

SWIFT

```
sdlManager.screenManager.presentSearchableChoiceSet(<#T##
choiceSet: SDLChoiceSet##SDLChoiceSet#>, mode: <#T##
SDLInteractionMode#>, with: <#T##SDLKeyboardDelegate#>)
```

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.

OBJECTIVE-C

```
[self.sdlManager.screenManager deleteChoices:<#(nonnull NSArray<SDLChoiceCell *> *)#>];
```

SWIFT

```
sdlManager.screenManager.deleteChoices(<#T##choices: [SDLChoiceCell]##[SDLChoiceCell]#>)
```

Presenting a Keyboard

Presenting a keyboard or a searchable menu requires you to additionally implement the `SDLKeyboardDelegate`. Note that the `initialText` in the keyboard case often acts as "placeholder text" *not* as true initial text.

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.

OBJECTIVE-C

```
[self.sdlManager.screenManager presentKeyboardWithInitialText:<#(
nonnull NSString *)#> delegate:<#(nonnull id<SDLKeyboardDelegate>
)#>];
```

SWIFT

```
sdlManager.screenManager.presentKeyboard(withInitialText: <#T##
String#>, delegate: <#T##SDLKeyboardDelegate#>)
```

Implementing the Keyboard Delegate

Using the `SDLKeyboardDelegate` 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.

OBJECTIVE-C

```
#pragma mark - SDLKeyboardDelegate

/// Required Methods
- (void)keyboardDidAbortWithReason:(SDLKeyboardEvent)event {
    if ([event isEqualToEnum:SDLKeyboardEventCancelled]) {
        <#The user cancelled the keyboard interaction#>
    } else if ([event isEqualToEnum:SDLKeyboardEventAborted]) {
        <#The system aborted the keyboard interaction#>
    }
}

- (void)userDidSubmitInput:(NSString *)inputText withEvent:(
SDLKeyboardEvent)source {
    if ([source isEqualToEnum:SDLKeyboardEventSubmitted]) {
        <#The user submitted some text with the keyboard#>
    } else if ([source isEqualToEnum:SDLKeyboardEventVoice]) {
        <#The user decided to start voice input, you should start an
AudioPassThru session if supported#>
    }
}

/// Optional Methods
- (void)updateAutocompleteWithInput:(NSString *)currentInputText
completionHandler:(SDLKeyboardAutocompleteCompletionHandler)
completionHandler {
    <#Check the input text and return a string with the current
autocomplete text#>
}

- (void)updateCharacterSetWithInput:(NSString *)currentInputText
completionHandler:(SDLKeyboardCharacterSetCompletionHandler)
completionHandler {
    <#Check the input text and return a set of characters to allow the
user to enter#>
}

- (void)keyboardDidSendEvent:(SDLKeyboardEvent)event text:(NSString
 *)currentInputText {
    <#This is sent upon every event, such as keypresses, cancellations,
and aborting#>
}

- (SDLKeyboardProperties *)customKeyboardConfiguration {
    <#Use an alternate keyboard configuration. The keypressMode,
limitedCharacterSet, and autoCompleteText will be overridden by the
screen manager#>
}
```

```
extension <#Class Name#>: SDLKeyboardDelegate {
    /// Required Methods
    func keyboardDidAbort(withReason event: SDLKeyboardEvent) {
        switch event {
        case .cancelled:
            <#The user cancelled the keyboard interaction#>
        case .aborted:
            <#The system aborted the keyboard interaction#>
        default: break
        }
    }

    func userDidSubmitInput(_ inputText: String, withEvent source:
SDLKeyboardEvent) {
        switch source {
        case .voice:
            <#The user decided to start voice input, you should start an
AudioPassThru session if supported#>
        case .submitted:
            <#The user submitted some text with the keyboard#>
        default: break
        }
    }

    /// Optional Methods
    func updateAutocomplete(withInput currentInputText: String,
completionHandler: @escaping
SDLKeyboardAutocompleteCompletionHandler) {
        <#Check the input text and return a string with the current
autocomplete text#>
    }

    func updateCharacterSet(withInput currentInputText: String,
completionHandler: @escaping
SDLKeyboardCharacterSetCompletionHandler) {
        <#Check the input text and return a set of characters to allow the
user to enter#>
    }

    func keyboardDidSendEvent(_ event: SDLKeyboardEvent, text
currentInputText: String) {
        <#This is sent upon every event, such as keypresses,
cancellations, and aborting#>
    }

    func customKeyboardConfiguration() -> SDLKeyboardProperties {
        <#Use an alternate keyboard configuration. The keypressMode,
limitedCharacterSet, and autoCompleteText will be overridden by the
screen manager#>
```

```
    }
}
```

Using RPCs

If you don't want to use the `SDLScreenManager`, you can do this manually using the `SDLChoice`, `SDLCreateInteractionChoiceSet`, and `SDLPerformInteraction` RPC requests. You will need to create `SDLChoice`s, bundle them into `SDLCreateInteractionChoiceSet`s, and then present those choice sets via `SDLPerformInteraction`. As this is no longer a recommended course of action, we will leave it to you to figure out how to manually do it.

Alerts

An alert is a pop-up window with some lines of text and optional soft 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 simply be ignored.

> NOTE
>
> The alert will persist on the screen until the timeout has elapsed, or the user dismisses the alert by selecting a button. There is no way to dismiss the alert programmatically other than to set the timeout length.

Alert UI

Depending 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.

Ford HMI

Ford HMI

Alert TTS

The alert can also be formatted to speak a prompt when the alert appears on the screen. Do this by setting the `ttsChunks` parameter. To play the alert tone before the text-to-speech is spoken, set `playTone` to `true`.

OBJECTIVE-C

```
SDLAlert *alert = [[SDLAlert alloc] initWithAlertText1:@"<#Line 1#>"
alertText2:@"<#Line 2#>" alertText3:@"<#Line 3#>"];

// Maximum time alert appears before being dismissed
// Timeouts are must be between 3-10 seconds
// Timeouts may not work when soft buttons are also used in the alert
alert.duration = @5000;

// A progress indicator (e.g. spinning wheel or hourglass)
// Not all head units support the progress indicator
alert.progressIndicator = @YES;

// Text-to-speech
alert.ttsChunks = [SDLTTSChunk textChunksFromString:@"<#Text to
speak#>"];

// Special tone played before the tts is spoken
alert.playTone = @YES;

// Soft buttons
SDLSoftButton *okButton = [[SDLSoftButton alloc] init];
okButton.text = @"OK";
okButton.type = SDLSoftButtonTypeText;
okButton.softButtonID = @<#Soft Button Id#>;
okButton.handler = ^(SDLOnButtonPress *_Nullable buttonPress,
SDLOnButtonEvent *_Nullable buttonEvent) {
    if (buttonPress == nil) {
      return;
    }

    // create a custom action for the selected button
};

alert.softButtons = @[okButton];

// Send the alert
[self.sdlManager sendRequest:alert withResponseHandler:^(
SDLRPCRequest *request, SDLRPCResponse *response, NSError *error)
{
    if ([response.resultCode isEqualToEnum:SDLResultSuccess]) {
      // alert was dismissed successfully
    }
}];
```

SWIFT

```swift
let alert = SDLAlert(alertText1: "<#Line 1#>", alertText2: "<#Line 2#>", alertText3: "<#Line 3#>")

// Maximum time alert appears before being dismissed
// Timeouts are must be between 3-10 seconds
// Timeouts may not work when soft buttons are also used in the alert
alert.duration = 5000

// A progress indicator (e.g. spinning wheel or hourglass)
// Not all head units support the progress indicator
alert.progressIndicator = true

// Text-to-speech
alert.ttsChunks = SDLTTSChunk.textChunks(from: "<#Text to speak#>")

// Special tone played before the tts is spoken
alert.playTone = true

// Soft buttons
let okButton = SDLSoftButton()
okButton.text = "OK"
okButton.type = .text
okButton.softButtonID = <#Soft Button Id#>
okButton.handler = { (buttonPress, buttonEvent) in
    guard let press = buttonPress else { return }

    // create a custom action for the selected button
}

alert.softButtons = [okButton]

// Send the alert
sdlManager.send(request: alert) { (request, response, error) in
    if response?.resultCode == .success {
        // alert was dismissed successfully
    }
}
```

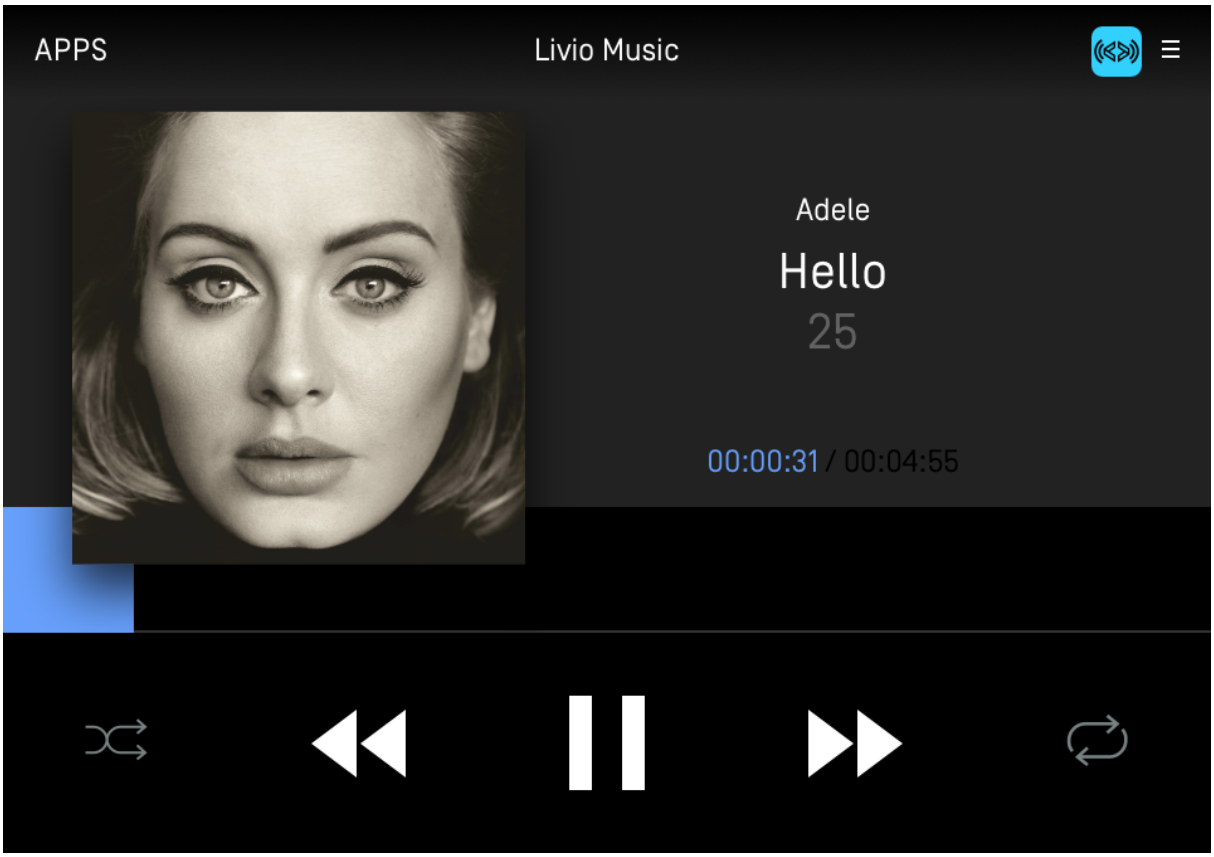
Media Clock

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. It is controlled via the `SetMediaClockTimer` RPC.

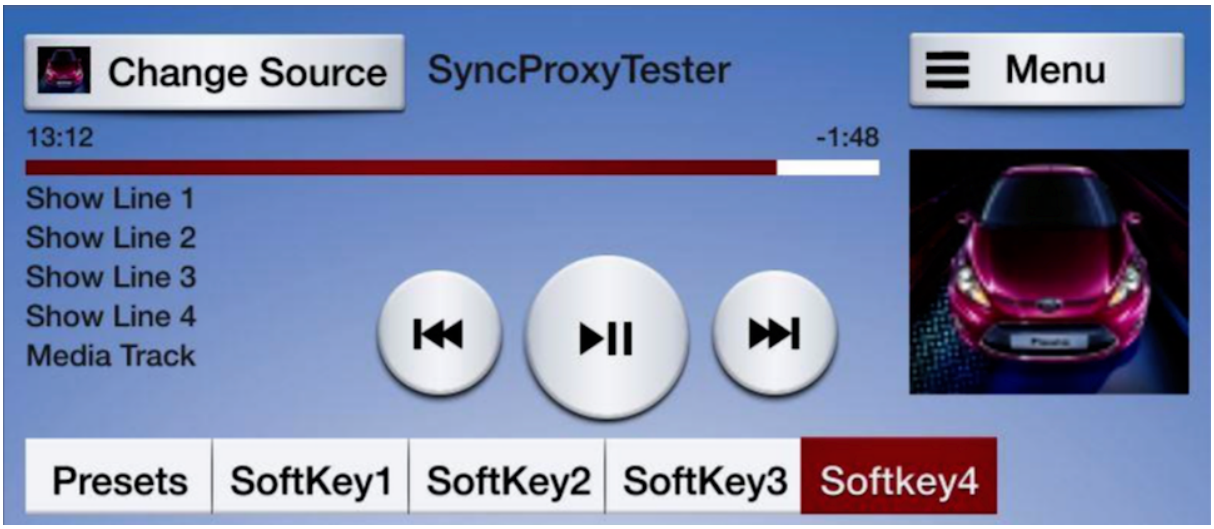
> NOTE
>
> Ensure your app is a media type app and you are using the media template before implementing this feature.

The media timer works like a timer. You set the start and end time and you set the timer to start ticking automatically either up or down.

Generic HMI



Ford HMI



Counting Up

In order to count up using the timer, the "bottom end" of the time will be 0:00, you can set the progress "start time", and the "top end". For example, if you are starting a song at `0:00` and it will end at `4:13`, you will end with a timer starting at `0:00`, ending at `0:00` with zero progress. This timer will automatically increment every second.

> NOTE
>
> The end time *must* be larger than the start time, or the request will be rejected

OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModeCountUp];mediaClock.startTime
= [[SDLStartTime alloc] initWithHours:0 minutes:0 seconds:0];
mediaClock.endTime = [[SDLStartTime alloc] initWithHours:0 minutes:4
 seconds:13];
[self.sdlManager sendRequest:mediaClock];
```

SWIFT

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .countUp)
mediaClock.startTime = SDLStartTime(hours: 0, minutes: 0, seconds: 0)
mediaClock.endTime = SDLStartTime(hours: 0, minutes: 4, seconds: 13
)
sdlManager.send(mediaClock)
```

The following code will create a timer starting at `0:00`, ending at `4:13` and already at `2:20` of progress.

OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModeCountUp];
mediaClock.startTime = [[SDLStartTime alloc] initWithHours:0 minutes:
2 seconds:20];
mediaClock.endTime = [[SDLStartTime alloc] initWithHours:0 minutes:4
 seconds:13];
[self.sdlManager sendRequest:mediaClock];
```

SWIFT

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .countUp)
mediaClock.startTime = SDLStartTime(hours: 0, minutes: 2, seconds:
20)
mediaClock.endTime = SDLStartTime(hours: 0, minutes: 4, seconds: 13
)
sdlManager.send(mediaClock)
```

Counting Down

Counting down is the opposite of counting up (I know, right?). The timer bar moves from right to left and the timer will count down. For example, if you're counting down from `10:00` to `0:00`, the timer will auto-update every second.

> NOTE
>
> The end time *must* be larger than the start time, or the request will be rejected

OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModeCountDown];
mediaClock.startTime = [[SDLStartTime alloc] initWithHours:0 minutes:
10 seconds:0];
mediaClock.endTime = [[SDLStartTime alloc] initWithHours:0 minutes:0
 seconds:0];
[self.sdlManager sendRequest:mediaClock];
```

SWIFT

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .countDown)
mediaClock.startTime = SDLStartTime(hours: 0, minutes: 10, seconds:
0)
mediaClock.endTime = SDLStartTime(hours: 0, minutes: 0, seconds: 0)
sdlManager.send(mediaClock)
```

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.

OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModePause];
[self.sdlManager sendRequest:mediaClock];
```

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModeResume];
[self.sdlManager sendRequest:mediaClock];
```

SWIFT

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .pause)
sdlManager.send(mediaClock)
```

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .resume)
sdlManager.send(mediaClock)
```

Clearing the Timer

Clearing the timer removes it from the screen.

OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModeClear];
[self.sdlManager sendRequest:mediaClock];
```

SWIFT

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .clear)
sdlManager.send(mediaClock)
```

Updating the Audio Indicator

The audio indicator is, essentially, the play / pause button. As of SDL v6.1, when connected to an SDL v5.0+ head unit, you can tell the system what icon to display on the play / pause button to correspond with how your app works.

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.

The code below will pause the media clock and set the audio indicator to show a play symbol.

OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [[SDLSetMediaClockTimer alloc
] initWithUpdateMode:SDLUpdateModePause];
mediaClock.audioStreamingIndicator =
SDLAudioStreamingIndicatorPlay;
[self.sdlManager sendRequest:mediaClock];
```

SWIFT

```
let mediaClock = SDLSetMediaClockTimer(updateMode: .pause)
mediaClock.audioStreamingIndicator = .play
sdlManager.send(mediaClock)
```

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 uploading a group of artworks. 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 there is no guarantee of the order in which the RPCs will be sent or in which order Core will return responses.

OBJECTIVE-C

```
SDLArtwork *artwork1 = [SDLArtwork artworkWithImage:[UIImage
imageNamed:@"<#Image name#>"] asImageFormat:<#
SDLArtworkImageFormat#>];
SDLArtwork *artwork2 = [SDLArtwork artworkWithImage:[UIImage
imageNamed:@"<#Image name#>"] asImageFormat:<#
SDLArtworkImageFormat#>];
[self.sdlManager sendRequests:@[artwork1, artwork2] progressHandler:
^(__kindof SDLRPCRequest * _Nonnull request, __kindof
SDLRPCResponse * _Nullable response, NSError * _Nullable error, float
percentComplete) {
    NSLog(@"Command %@ sent %@, percent complete %f%%",
request.name, response.resultCode == SDLResultSuccess ?
@"successfully" : @"unsuccessfully", percentComplete * 100);
} completionHandler:^(BOOL success) {
    NSLog(@"All requests sent %@", success ? @"successfully" :
@"unsuccessfully");
}];
```

SWIFT

```
let artwork1 = SDLArtwork(image: <#UIImage#>, persistent: <#Bool#
>, as: <#SDLArtworkImageFormat#>)
let artwork2 = SDLArtwork(image: <#UIImage#>, persistent: <#Bool#
>, as: <#SDLArtworkImageFormat#>)
sdlManager.send([artwork1, artwork2], progressHandler: { (request,
response, error, percentComplete) in
    print("Command \(request.name) sent \(response?.resultCode == .
success ? "successfully" : "unsuccessfully"), percent complete \(
percentComplete * 100)")
}) { success in
    print("All requests sent \(success ? "successfully" : "unsuccessfully")"
)
}
```

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.

OBJECTIVE-C

```
SDLChoice *choice = [[SDLChoice alloc] initWithId:<#Choice Id#>
menuName:@"<#Menu Name#>" vrCommands:@[@"<#VR
Command#>"]];
SDLCreateInteractionChoiceSet *createInteractionChoiceSet = [[
SDLCreateInteractionChoiceSet alloc] initWithId:<#Choice Set Id#>
choiceSet:@[choice]];
SDLPerformInteraction *performInteraction = [[SDLPerformInteraction
alloc] initWithInteractionChoiceSetId:<#Choice Set Id#>];

[self.sdlManager sendSequentialRequests:@[
createInteractionChoiceSet, performInteraction] progressHandler:^
BOOL(__kindof SDLRPCRequest * _Nonnull request, __kindof
SDLRPCResponse * _Nullable response, NSError * _Nullable error, float
percentComplete) {
    NSLog(@"Command %@ sent %@, percent complete %f%%",
request.name, response.resultCode == SDLResultSuccess ?
@"successfully" : @"unsuccessfully", percentComplete * 100);
} completionHandler:^(BOOL success) {
    NSLog(@"All requests sent %@", success ? @"successfully" :
@"unsuccessfully");
}];
```

SWIFT

```
let choice = SDLChoice(id: <#Choice Id#>, menuName: "<#Menu
Name#>", vrCommands: ["<#VR Command#>"])
let createInteractionChoiceSet = SDLCreateInteractionChoiceSet(id: <#
Choice Set Id#>, choiceSet: [choice])
let performInteraction = SDLPerformInteraction(interactionChoiceSetId:
<#Choice Set Id#>)

sdlManager.sendSequential(requests: [createInteractionChoiceSet,
performInteraction], progressHandler: { (request, response, error,
percentageCompleted) -> Bool in
    print("Command \(request.name) sent \(response?.resultCode == .
success ? "successfully" : "unsuccessfully"), percent complete \(
percentComplete * 100)")
}) { success in
    print("All requests sent \(success ? "successfully" : "unsuccessfully")"
)
}
```

Retrieving Vehicle Data

Use the `SDLGetVehicleData` RPC call to get vehicle data. The HMI level must be `FULL`, `LIMITED`, or `BACKGROUND` in order to get data.

Each vehicle manufacturer decides which data it will expose and to whom they will expose it. Please check the response to find out which data you will have access to in your head unit. Additionally, be aware the the driver / user may have the ability to disable vehicle data through the settings menu of their infotainment head unit.

> NOTE
>
> You may only ask for vehicle data that is available to your `appName` & `appId` combination. These will be specified by each OEM separately. See Understanding Permissions for more details.

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
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. The Ignition status. The ignition stable status. The park brake active status.
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
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
E-Call Infomation	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

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

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
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
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
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
Speed	speed	Speed in KPH
Steering Wheel Angle	steeringWheelAngle	Current angle of the steering wheel (in degrees)

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
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
RPM	rpm	The number of revolutions per minute of the engine
VIN	vin	The Vehicle Identification Number
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

Using `SDLGetVehicleData`, we can ask for vehicle data a single time, if needed.

OBJECTIVE-C

```
SDLGetVehicleData *getVehicleData = [[SDLGetVehicleData alloc] init];
getVehicleData.prndl = @YES;
[self.sdlManager sendRequest:getVehicleData withResponseHandler:^(
__kindof SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse
 * _Nullable response, NSError * _Nullable error) {
    if (error || ![response isKindOfClass:SDLGetVehicleDataResponse.
class]) {
        NSLog(@"Encountered Error sending GetVehicleData: %@", error);
        return;
    }

    SDLGetVehicleDataResponse* getVehicleDataResponse = (
SDLGetVehicleDataResponse *)response;
    SDLResult *resultCode = getVehicleDataResponse.resultCode;
    if (![resultCode isEqualToEnum:SDLResultSuccess]) {
        if ([resultCode isEqualToEnum:SDLResultRejected]) {
            NSLog(@"GetVehicleData was rejected. Are you in an
appropriate HMI?");
        } else if ([resultCode isEqualToEnum:SDLResultDisallowed]) {
            NSLog(@"Your app is not allowed to use GetVehicleData");
        } else {
            NSLog(@"Some unknown error has occured!");
        }
        return;
    }

    SDLPRNDL *prndl = getVehicleDataResponse.prndl;
}];
```

SWIFT

```swift
let getVehicleData = SDLGetVehicleData()
getVehicleData.prndl = true
sdlManager.send(getVehicleData) { (request, response, error) in
    guard let response = response as? SDLGetVehicleDataResponse else
{ return }

    if let error = error {
        print("Encountered Error sending GetVehicleData: \(error)")
        return
    }

    if !response.resultCode == .success {
        if response.resultCode == .rejected {
            print("GetVehicleData was rejected. Are you in an appropriate
HMI?")
        } else if response.resultCode == .disallowed {
            print("Your app is not allowed to use GetVehicleData")
        } else {
            print("Some unknown error has occured!")
        }
        return
    }

    guard let prndl = response.prndl else { return }
}
```

Subscribing to Vehicle Data

Subscribing to vehicle data allows you to get notified whenever we have new data available. This data should not be relied upon being received in a consistent manner. New vehicle data is available roughly every second.

First, register to observe the `SDLDidReceiveVehicleDataNotification` notification:

OBJECTIVE-C

```
[[NSNotificationCenter defaultCenter] addObserver:self selector:
@selector(vehicleDataAvailable:) name:
SDLDidReceiveVehicleDataNotification object:nil];
```

SWIFT

```
NotificationCenter.default.addObserver(self, selector: #selector(
vehicleDataAvailable(_:)), name: .SDLDidReceiveVehicleData, object: nil
)
```

Then send the Subscribe Vehicle Data Request:

OBJECTIVE-C

```
SDLSubscribeVehicleData *subscribeVehicleData = [[
SDLSubscribeVehicleData alloc] init];
subscribeVehicleData.prndl = @YES;

[self.sdlManager sendRequest:subscribeVehicleData
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
__kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
error) {
    if (![response isKindOfClass:[SDLSubscribeVehicleDataResponse class
]]) {
        return;
    }

    SDLSubscribeVehicleDataResponse *subscribeVehicleDataResponse
= (SDLSubscribeVehicleDataResponse*)response;
    SDLVehicleDataResult *prndlData = subscribeVehicleDataResponse.
prndl;
    if (!response.success.boolValue) {
        if ([response.resultCode isEqualToEnum:SDLResultDisallowed]) {
            // Not allowed to register for this vehicle data.
        } else if ([response.resultCode isEqualToEnum:
SDLResultUserDisallowed]) {
            // User disabled the ability to give you this vehicle data
        } else if ([response.resultCode isEqualToEnum:SDLResultIgnored])
{
            if ([prndlData.resultCode isEqualToEnum:
SDLVehicleDataResultCodeDataAlreadySubscribed]) {
                // You have access to this data item, and you are already
subscribed to this item so we are ignoring.
            } else if ([prndlData.resultCode isEqualToEnum:
SDLVehicleDataResultCodeVehicleDataNotAvailable]) {
                // You have access to this data item, but the vehicle you are
connected to does not provide it.
            } else {
                NSLog(@"Unknown reason for being ignored: %@", prndlData
.resultCode.value);
            }
        } else if (error) {
            NSLog(@"Encountered Error sending SubscribeVehicleData: %
@", error);
        }

        return;
    }

    // Successfully subscribed
}];
```

SWIFT

```swift
let subscribeVehicleData = SDLSubscribeVehicleData()
subscribeVehicleData.prndl = true

sdlManager.send(request: subscribeVehicleData) { (request, response, error) in
    guard let response = response as? SDLSubscribeVehicleDataResponse else { return }

    guard response.success.boolValue == true else {
        if response.resultCode == .disallowed {
            // Not allowed to register for this vehicle data.
        } else if response.resultCode == .userDisallowed {
            // User disabled the ability to give you this vehicle data
        } else if response.resultCode == .ignored {
            if let prndlData = response.prndl {
                if prndlData.resultCode == .dataAlreadySubscribed {
                    // You have access to this data item, and you are already subscribed to this item so we are ignoring.
                } else if prndlData.resultCode == .vehicleDataNotAvailable {
                    // You have access to this data item, but the vehicle you are connected to does not provide it.
                } else {
                    print("Unknown reason for being ignored: \(prndlData.resultCode)")
                }
            } else {
                print("Unknown reason for being ignored: \(String(describing: response.info))")
            }
        } else if let error = error {
            print("Encountered Error sending SubscribeVehicleData: \(error)")
        }
        return
    }

    // Successfully subscribed
}
```

Finally, react to the notification when Vehicle Data is received:

OBJECTIVE-C

```
- (void)vehicleDataAvailable:(SDLRPCNotificationNotification *)
notification {
    if (![notification.notification isKindOfClass:SDLOnVehicleData.class]) {
        return;
    }

    SDLOnVehicleData *onVehicleData = (SDLOnVehicleData *)
notification.notification;

    SDLPRNDL *prndl = onVehicleData.prndl;
}
```

SWIFT

```
func vehicleDataAvailable(_ notification: SDLRPCNotificationNotification
) {
    guard let onVehicleData = notification.notification as?
SDLOnVehicleData else {
        return
    }

    let prndl = onVehicleData.prndl
}
```

Unsubscribing from Vehicle Data

Sometimes you may not always need all of the vehicle data you are listening to. We suggest that you only are subscribing when the vehicle data is needed. To stop listening to specific vehicle data items, utilize `SDLUnsubscribeVehicleData`.

OBJECTIVE-C

```
SDLUnsubscribeVehicleData *unsubscribeVehicleData = [[
SDLUnsubscribeVehicleData alloc] init];
unsubscribeVehicleData.prndl = @YES;

[self.sdlManager sendRequest:unsubscribeVehicleData
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
__kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
error) {
    if (![response isKindOfClass:[SDLUnsubscribeVehicleDataResponse
class]]) {
        return;
    }

    SDLUnsubscribeVehicleDataResponse *
unsubscribeVehicleDataResponse = (
SDLUnsubscribeVehicleDataResponse*)response;
    SDLVehicleDataResult *prndlData =
unsubscribeVehicleDataResponse.prndl;

    if (!response.success.boolValue) {
        if ([response.resultCode isEqualToEnum:SDLResultDisallowed]) {
            // Not allowed to register for this vehicle data, so unsubscribe
also will not work.
        } else if ([response.resultCode isEqualToEnum:
SDLResultUserDisallowed]) {
            // User disabled the ability to give you this vehicle data, so
unsubscribe also will not work.
        } else if ([response.resultCode isEqualToEnum:SDLResultIgnored])
{
            if ([prndlData.resultCode isEqualToEnum:
SDLVehicleDataResultCodeDataNotSubscribed]) {
                // You have access to this data item, but it was never
subscribed to so we ignored it.
            } else {
                NSLog(@"Unknown reason for being ignored: %@", prndlData
.resultCode.value);
            }
        } else if (error) {
            NSLog(@"Encountered Error sending UnsubscribeVehicleData:
%@", error);
        }
        return;
    }

    // Successfully unsubscribed
}];
```

SWIFT

```swift
let unsubscribeVehicleData = SDLUnsubscribeVehicleData()
unsubscribeVehicleData.prndl = true

sdlManager.send(request: unsubscribeVehicleData) { (request,
response, error) in
    guard let response = response as?
SDLUnsubscribeVehicleDataResponse else { return }

    guard response.success.boolValue == true else {
        if response.resultCode == .disallowed {

        } else if response.resultCode == .userDisallowed {

        } else if response.resultCode == .ignored {
            if let prndlData = response.prndl {
                if prndlData.resultCode == .dataNotSubscribed {
                    // You have access to this data item, and you are already
unsubscribed to this item so we are ignoring.
                } else if prndlData.resultCode == .vehicleDataNotAvailable {
                    // You have access to this data item, but the vehicle you
are connected to does not provide it.
                } else {
                    print("Unknown reason for being ignored: \(prndlData.
resultCode)")
                }
            } else {
                print("Unknown reason for being ignored: \(String(describing:
response.info))")
            }
        } else if let error = error {
            print("Encountered Error sending UnsubscribeVehicleData: \(
error)")
        }
        return
    }

    // Successfully unsubscribed
}
```

Calling a Phone Number

Dialing a Phone Number allows you to send a phone number to dial on the user's phone. Regardless of platform (Android or iOS), you must be sure that a device is connected via Bluetooth (even if using iOS/USB) for this RPC to work. If it is not connected, you will receive a REJECTED `resultCode`.

> NOTE
>
> DialNumber is an RPC that is usually restricted by OEMs. As a result, the OEM you are connecting to may limit app functionality if not approved for usage.

Detecting if DialNumber is Available

`DialNumber` is a newer RPC, so there is a possibility that not all head units will support it. To see if `DialNumber` is supported, you may look at `SDLManager`'s `systemCapabilityManager.hmiCapabilities.phoneCall` property after the ready handler is called.

OBJECTIVE-C

```
BOOL isPhoneCallSupported = NO;

[self.sdlManager startWithReadyHandler:^(BOOL success, NSError *
_Nullable error) {
    if (!success) {
        NSLog(@"SDL errored starting up: %@", error);
        return;
    }

    SDLHMICapabilities *hmiCapabilities = self.sdlManager.
systemCapabilityManager.hmiCapabilities;
    if (hmiCapabilities != nil) {
        isPhoneCallSupported = hmiCapabilities.phoneCall.boolValue;
    }
}];
```

SWIFT

```
var isPhoneCallSupported = false

sdlManager.start { (success, error) in
    if !success {
        print("SDL errored starting up: \(error.debugDescription)")
        return
    }

    if let hmiCapabilities = self.sdlManager.systemCapabilityManager.
hmiCapabilities, let phoneCallsSupported = hmiCapabilities.phoneCall?.
boolValue {
        isPhoneCallSupported = phoneCallsSupported
    }
}
```

Sending a DialNumber Request

> NOTE
>
> For DialNumber, all characters are stripped except for `0`-`9`, `*`, `#`, `,`, `;`, and `+`

OBJECTIVE-C

```
SDLDialNumber *dialNumber = [[SDLDialNumber alloc] init];
dialNumber.number = @"1238675309";

[self.sdlManager sendRequest:dialNumber withResponseHandler:^(
__kindof SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse
 * _Nullable response, NSError * _Nullable error) {
    if (error != nil || ![response isKindOfClass:SDLDialNumberResponse.
class]) {
        NSLog(@"Encountered Error sending DialNumber: %@", error);
        return;
    }

    SDLDialNumberResponse* dialNumber = (SDLDialNumberResponse *
)response;
    SDLResult *resultCode = dialNumber.resultCode;
    if (![resultCode isEqualToEnum:SDLResultSuccess]) {
        if ([resultCode isEqualToEnum:SDLResultRejected]) {
            NSLog(@"DialNumber was rejected. Either the call was sent
and cancelled or there is no device connected");
        } else if ([resultCode isEqualToEnum:SDLResultDisallowed]) {
            NSLog(@"Your app is not allowed to use DialNumber");
        } else {
            NSLog(@"Some unknown error has occured!");
        }
        return;
    }

    // Successfully sent!
}];
```

SWIFT

```
let dialNumber = SDLDialNumber()
dialNumber.number = "1238675309"

sdlManager.send(request: dialNumber) { (request, response, error) in
    guard let response = response as? SDLDialNumberResponse else {
return }

    if let error = error {
        print("Encountered Error sending DialNumber: \(error)")
        return
    }

    if response.resultCode != .success {
        if response.resultCode == .rejected {
            print("DialNumber was rejected. Either the call was sent and
cancelled or there is no device connected")
        } else if response.resultCode == .disallowed {
            print("Your app is not allowed to use DialNumber")
        } else {
            print("Some unknown error has occured!")
        }
        return
    }

    // Successfully sent!
}
```

DialNumber Result

`DialNumber` has 3 possible results that you should expect:

1. SUCCESS - DialNumber was successfully sent, and a phone call was initiated by the user.
2. REJECTED - DialNumber was sent, and a phone call was cancelled by the user. Also, this could mean that there is no phone connected via Bluetooth.
3. DISALLOWED - Your app does not have permission to use DialNumber.

Setting the Navigation Destination

Setting a Navigation Destination allows you to send a GPS location, prompting the user to navigate to that location using their embedded navigation. When using the `SendLocation` RPC, you will not receive a callback about how the user interacted with this location, only if it was successfully sent to Core and received. It will be handled by Core from that point on using the embedded navigation system.

> NOTE
>
> This currently is only supported for Embedded Navigation; it does not work with Mobile Navigation Apps at this time.

Detecting if SendLocation is Available

To check if `SendLocation` is supported, you may look at `SDLManager`'s `systemCapabilityManager` property after the ready handler is called. Or, you may use `SDLManager`'s `permissionManager` property to ask for the permission status of `SendLocation`.

NOTE

`SendLocation` is an RPC that is usually restricted by OEMs. As a result, the OEM you are connecting to may limit app functionality if you are not approved to use it.

OBJECTIVE-C

```
BOOL isNavigationSupported = NO;

__weak typeof (self) weakSelf = self;
[self.sdlManager startWithReadyHandler:^(BOOL success, NSError *
_Nullable error) {
    if (!success) {
        NSLog(@"SDL errored starting up: %@", error);
        return;
    }

    SDLHMICapabilities *hmiCapabilities = self.sdlManager.
systemCapabilityManager.hmiCapabilities;
    if (hmiCapabilities != nil) {
        isNavigationSupported = hmiCapabilities.navigation.boolValue;
    }
}];
```

SWIFT

```
var isNavigationSupported = false

sdlManager.start { (success, error) in
    if !success {
        print("SDL errored starting up: \(error.debugDescription)")
        return
    }

    if let hmiCapabilities = self.sdlManager.systemCapabilityManager.
hmiCapabilities, let navigationSupported = hmiCapabilities.navigation?.
boolValue {
        isNavigationSupported = navigationSupported
    }
}
```

Using Send Location

To use `SendLocation`, you must at least include the Longitude and Latitude of the location.

OBJECTIVE-C

```
SDLSendLocation *sendLocation = [[SDLSendLocation alloc]
initWithLongitude:-97.380967 latitude:42.877737 locationName:@"The
Center" locationDescription:@"Center of the United States" address:@[
@"900 Whiting Dr", @"Yankton, SD 57078"] phoneNumber:nil image:nil
];
[self.sdlManager sendRequest:sendLocation withResponseHandler:^(
__kindof SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse
 * _Nullable response, NSError * _Nullable error) {
    if (error || ![response isKindOfClass:SDLSendLocationResponse.class])
 {
        NSLog(@"Encountered Error sending SendLocation: %@", error);
        return;
    }

    SDLSendLocationResponse *sendLocation = (
SDLSendLocationResponse *)response;
    SDLResult *resultCode = sendLocation.resultCode;
    if (![resultCode isEqualToEnum:SDLResultSuccess]) {
        if ([resultCode isEqualToEnum:SDLResultInvalidData]) {
            NSLog(@"SendLocation was rejected. The request contained
invalid data.");
        } else if ([resultCode isEqualToEnum:SDLResultDisallowed]) {
            NSLog(@"Your app is not allowed to use SendLocation");
        } else {
            NSLog(@"Some unknown error has occured!");
        }
        return;
    }

    // Successfully sent!
}];
```

SWIFT

```swift
let sendLocation = SDLSendLocation(longitude: -97.380967, latitude:
42.877737, locationName: "The Center", locationDescription: "Center
of the United States", address: ["900 Whiting Dr", "Yankton, SD 57078"
], phoneNumber: nil, image: nil)

sdlManager.send(request: sendLocation) { (request, response, error) in
    guard let response = response as? SDLSendLocationResponse else {
return }

    if let error = error {
        print("Encountered Error sending SendLocation: \(error)")
        return
    }

    if response.resultCode != .success {
        if response.resultCode == .invalidData {
            print("SendLocation was rejected. The request contained
invalid data.")
        } else if response.resultCode == .disallowed {
            print("Your app is not allowed to use SendLocation")
        } else {
            print("Some unknown error has occured!")
        }
        return
    }

    // Successfully sent!
}
```

Determining the Result of SendLocation

`SendLocation` has 3 possible results that you should expect:

1. SUCCESS - SendLocation was successfully sent.
2. INVALID_DATA - The request you sent contains invalid data and was rejected.
3. DISALLOWED - Your app does not have permission to use SendLocation.

In-Car 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, we must leverage the `SDLPerformAudioPassThru` RPC.

> NOTE
>
> PerformAudioPassThru 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`.

In order to know the currently supported audio capture capabilities of the connected head unit, please refer to the `SDLSystemCapabilityManager.audioPassThruCapabilities` documentation.

> 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.

Starting Audio Capture

To initiate audio capture, we must construct a `SDLPerformAudioPassThru` request. The properties we will set in this object's constructor relate to how we wish to gather the audio data from the vehicle we are connected to.

> NOTE
>
> Currently, SDL only supports Sampling Rates of 16 khz and Bit Rates of 16 bit.

OBJECTIVE-C

```
SDLPerformAudioPassThru *audioPassThru = [[
SDLPerformAudioPassThru alloc] initWithInitialPrompt:@"<#A speech
prompt when the dialog appears#>" audioPassThruDisplayText1:
@"<#Ask me \"What's the weather?\"#>" audioPassThruDisplayText2:
@"<#or \"What is 1 + 2?\"#>" samplingRate:SDLSamplingRate16KHZ
bitsPerSample:SDLBitsPerSample16Bit audioType:SDLAudioTypePCM
maxDuration:<#Time in milliseconds to keep the dialog open#>
muteAudio:YES];

[self.sdlManager sendRequest:audioPassThru];
```

SWIFT

```
let audioPassThru = SDLPerformAudioPassThru(initialPrompt: "<#A
speech prompt when the dialog appears#>",
audioPassThruDisplayText1: "<#Ask me \"What's the weather?\"#>",
audioPassThruDisplayText2: "<#or \"What is 1 + 2?\"#>",
samplingRate: .rate16KHZ, bitsPerSample: .sample16Bit, audioType: .
PCM, maxDuration: <#Time in milliseconds to keep the dialog open#>,
 muteAudio: true)

sdlManager.send(audioPassThru)
```

Ford HMI

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 developer must be in charge of managing previously retrieved audio data.

OBJECTIVE-C

```
SDLPerformAudioPassThru *audioPassThru = [[
SDLPerformAudioPassThru alloc] initWithInitialPrompt:@"<#A speech
prompt when the dialog appears#>" audioPassThruDisplayText1:
@"<#Ask me \"What's the weather?\"#>" audioPassThruDisplayText2:
@"<#or \"What is 1 + 2?\"#>" samplingRate:SDLSamplingRate16KHZ
bitsPerSample:SDLBitsPerSample16Bit audioType:SDLAudioTypePCM
maxDuration:<#Time in milliseconds to keep the dialog open#>
muteAudio:YES];

audioPassThru.audioDataHandler = ^(NSData * _Nullable audioData) {
    // Do something with current audio data.
    if (audioData.length == 0) { return; }
    <#code#>
}

[self.sdlManager sendRequest:audioPassThru];
```

SWIFT

```swift
let audioPassThru = SDLPerformAudioPassThru(initialPrompt: "<#A
speech prompt when the dialog appears#>",
audioPassThruDisplayText1: "<#Ask me \"What's the weather?\"#>",
audioPassThruDisplayText2: "<#or \"What is 1 + 2?\"#>",
samplingRate: .rate16KHZ, bitsPerSample: .sample16Bit, audioType: .
PCM, maxDuration: <#Time in milliseconds to keep the dialog open#>,
 muteAudio: true)

audioPassThru.audioDataHandler = { (data) in
    // Do something with current audio data.
    guard let audioData = data else { return }
    <#code#>
}

sdlManager.send(audioPassThru)
```

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

Perform Audio Pass Thru 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 PerformAudioPassThru is ended.

Audio Capture can be ended in 4 ways:

1. Audio Pass Thru has timed out.

   If the Audio Pass Thru has proceeded longer than the requested timeout duration, Core will end this request with a `resultCode` of `SUCCESS`. You should expect to handle this Audio Pass Thru as though it was successful.
2. Audio Pass Thru was closed due to user pressing "Cancel".

   If the Audio Pass Thru was displayed, and the user pressed the "Cancel" button, you will receive a `resultCode` of `ABORTED`. You should expect to ignore this Audio Pass Thru.
3. Audio Pass Thru was closed due to user pressing "Done".

   If the Audio Pass Thru was displayed, and the user pressed the "Done" button, you will receive a `resultCode` of `SUCCESS`. You should expect to handle this Audio Pass Thru as though it was successful.
4. Audio Pass Thru was ended due to the developer ending the request.

   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.

OBJECTIVE-C

```
SDLEndAudioPassThru *endAudioPassThru = [[SDLEndAudioPassThru alloc] init];
[self.sdlManager sendRequest:endAudioPassThru];
```

SWIFT

```
let endAudioPassThru = SDLEndAudioPassThru()
sdlManager.send(endAudioPassThru)
```

You will receive a `resultCode` of `SUCCESS`, and should expect to handle this audio pass thru as though it was successful.

Handling the Response

To process the response that we received from an ended audio capture, we use the `withResponseHandler` property in `SDLManager`'s `send(_ :)` function.

OBJECTIVE-C

```
[self.sdlManager sendRequest:performAudioPassThru
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
__kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
error) {
    if (error || ![response isKindOfClass:
SDLPerformAudioPassThruResponse.class]) {
        NSLog(@"Encountered Error sending Perform Audio Pass Thru: %
@", error);
        return;
    }

    SDLPerformAudioPassThruResponse *audioPassThruResponse = (
SDLPerformAudioPassThruResponse *)response;
    SDLResult *resultCode = audioPassThruResponse.resultCode;
    if (![resultCode isEqualToEnum:SDLResultSuccess]) {
        // Cancel any usage of the audio data
    }

    // Process audio data
}];
```

SWIFT

```
sdlManager.send(request: performAudioPassThru) { (request, response,
 error) in
    guard let response = response else { return }

    guard response.resultCode == .success else {
        // Cancel any usage of the audio data.
        return
    }

    // Process audio data
}
```

Uploading Files

In almost all cases, graphics are uploaded using the `ScreenManager`. You can find out about setting images in templates, soft buttons, and menus in the Text Images and Buttons guide. Other situations, such as `PerformInteraction`s, VR help lists, and turn by turn directions, are not currently covered by the `ScreenManager`. To upload an image, see the Uploading Images guide.

Uploading an mp3 Using SDLFileManager

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` uses a `UIImage`.

OBJECTIVE-C

```
NSData *mp3Data = <#Get the File Data#>;
SDLFile *file = [SDLFile fileWithData:mp3Data name:<#File name to be
referenced later#> fileExtension:<#File Extension#>];

[self.sdlManager.fileManager uploadFile:file completionHandler:^(BOOL
 success, NSUInteger bytesAvailable, NSError * _Nullable error) {
    if (error != nil) { return; }
    <#File Upload Successful#>
}];]
```

SWIFT

```
let mp3Data = <#Get MP3 Data#>
let file = SDLFile(data: mp3Data, name: <#File name#> fileExtension:
<#File Extension#>)

sdlManager.fileManager.upload(file: file) { (success, bytesAvailable,
error) in
    guard error == nil else { return }
    <#File Upload Successful#>
}
```

Batch 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.

OBJECTIVE-C

```
SDLFile *file1 = [SDLFile fileWithData:<#Data#> name:<#File name to
 be referenced later#> fileExtension:<#File Extension#>];
SDLFile *file2 = [SDLFile fileWithData:<#Data#> name:<#File name to
 be referenced later#> fileExtension:<#File Extension#>];

[self.sdlManager.fileManager uploadFiles:@[file1, file2] progressHandler
:^BOOL(NSString * _Nonnull fileName, float uploadPercentage, NSError
* _Nullable error) {
    // A single file has finished uploading. Use this to check for individual
errors, to use an file as soon as its uploaded, or to check the progress
of the upload
    // The upload percentage is calculated as the total file size of all
attempted file uploads (regardless of the successfulness of the upload)
divided by the sum of the data in all the files
    // Return YES to continue sending files. Return NO to cancel any files
that have not yet been sent.
} completionHandler:^(NSArray<NSString *> * _Nonnull fileNames,
NSError * _Nullable error) {
    // All files have completed uploading.
    // If all files were uploaded successfully, the error will be nil
    // The error's userInfo parameter is of type [fileName: error message]
}];
```

SWIFT

```
let file1 = SDLFile(data: <#File Data#>, name: <#File name#>
fileExtension: <#File Extension#>)
let file2 = SDLFile(data: <#File Data#>, name: <#File name#>
fileExtension: <#File Extension#>)

sdlManager.fileManager.upload(files: [file1, file2], progressHandler: { (
fileName, uploadPercentage, error) -> Bool in
    // A single file has finished uploading. Use this to check for individual
errors, to use an file as soon as its uploaded, or to check the progress
of the upload
    // The upload percentage is calculated as the total file size of all
attempted file uploads (regardless of the successfulness of the upload)
divided by the sum of the data in all the files
    // Return true to continue sending files. Return false to cancel any
files that have not yet been sent.
}) { (fileNames, error) in
    // All files have completed uploading.
    // If all files were uploaded successfully, the error will be nil
    // The error's userInfo parameter is of type [fileName: error message]
}
```

File Persistance

`SDLFile` and its subclass `SDLArtwork` support uploading persistant files, i.e. files that are not deleted when the car turns off. Persistance 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 persistant because it will take up unnecessary space on the head unit. You can check the persistence via:

OBJECTIVE-C

```
if (file.isPersistent) {
    <#File was initialized as persistent#>
}
```

SWIFT

```
if file.isPersistent {
    <#File was initialized as persistent#>
}
```

NOTE

Be aware that persistance will not work if space on the head unit is limited. `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.

OBJECTIVE-C

```
file.overwrite = YES;
```

SWIFT

```
file.overwrite = true
```

Checking the Amount of File Storage

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

OBJECTIVE-C

```
NSUInteger bytesAvailable = self.sdlManager.fileManager.bytesAvailable;
```

SWIFT

```
let bytesAvailable = sdlManager.fileManager.bytesAvailable
```

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.

OBJECTIVE-C

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

SWIFT

```
if let fileIsOnHeadUnit = sdlManager.fileManager.remoteFileNames.
contains("<#Name Uploaded As#>") {
    if fileIsOnHeadUnit {
        <#File exists#>
    } else {
        <#File does not exist#>
    }
}
```

Deleting Stored Files

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

OBJECTIVE-C

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

SWIFT

```
sdlManager.fileManager.delete(fileName: "<#Save As Name#>") { (
success, bytesAvailable, error) in
    if success {
        <#Image was deleted successfully#>
    }

}
```

Batch Deleting Files

OBJECTIVE-C

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

SWIFT

```
sdlManager.fileManager.delete(fileNames: ["<#Save As Name#>",
"<#Save as Name 2#>"]) { (error) in
    if (error == nil) {
        <#Images were deleted successfully#>
    }
}
```

Uploading Images

> NOTE
>
> If you are looking to upload images for use in template graphics, soft buttons, or the menu, you can use the ScreenManager. Other situations, such as VR help lists and turn by turn directions, are not currently covered by the `ScreenManager`.

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. Persistant 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.

To learn how to use images once they are uploaded, please see Text, Images, and Buttons.

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 unneccessary image data. To check if graphics are supported look at the `SDLManager`'s `registerResponse` property once the `SDLManager` has started successfully.

OBJECTIVE-C

```
__weak typeof (self) weakSelf = self;
[self.sdlManager startWithReadyHandler:^(BOOL success, NSError *
_Nullable error) {
    if (!success) {
        NSLog(@"SDL errored starting up: %@", error);
        return;
    }

    SDLDisplayCapabilities *displayCapabilities = weakSelf.sdlManager.
registerResponse.displayCapabilities;
    BOOL areGraphicsSupported = NO;
    if (displayCapabilities != nil) {
        areGraphicsSupported = displayCapabilities.graphicSupported.
boolValue;
    }
}];
```

SWIFT

```
sdlManager.start { [weak self] (success, error) in
    if !success {
        print("SDL errored starting up: \(error.debugDescription)")
        return
    }

    var areGraphicsSupported = false
    if let displayCapabilities = self?.sdlManager.registerResponse?.
displayCapabilities {
        areGraphicsSupported = displayCapabilities.graphicSupported.
boolValue
    }
}
```

Uploading an Image Using SDLFileManager

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`.

OBJECTIVE-C

```
UIImage* image = [UIImage imageNamed:@"<#Image Name#>"];
if (!image) {
    <#Error Reading from Assets#>
    return;
}

SDLArtwork* artwork = [SDLArtwork artworkWithImage:image
asImageFormat:<#SDLArtworkImageFormat#>];

[self.sdlManager.fileManager uploadArtwork:artwork completionHandler
:^(BOOL success, NSString * _Nonnull artworkName, NSUInteger
bytesAvailable, NSError * _Nullable error) {
    if (error != nil) { return; }
    <#Image Upload Successful#>
    // To send the image as part of a show request, create a SDLImage
object using the artworkName
    SDLImage *image = [[SDLImage alloc] initWithName:artworkName];
}];
```

SWIFT

```
guard let image = UIImage(named: "<#Image Name#>") else {
    <#Error Reading from Assets#>
    return
}
let artwork = SDLArtwork(image: image, persistent: <#Bool#>, as: <#
SDLArtworkImageFormat#>)

sdlManager.fileManager.upload(artwork: artwork) { (success,
artworkName, bytesAvailable, error) in
    guard error == nil else { return }
    <#Image Upload Successful#>
    // To send the image as part of a show request, create a SDLImage
object using the artworkName
    let graphic = SDLImage(name: artworkName)
}
```

Batch File Uploads, Persistence, etc.

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

Playing Audio Indications

As of SDL v6.1, you can pass an uploaded Audio File's name to `TTSChunk`, allowing any API that takes a `TTSChunk` to pass and play your audio file. This can be used, for example, to play a distinctive audio chime or indication unique to your application, letting the user know that something has occurred. A sports app, for example, could use this to notify the user of a score update alongside an `Alert` request.

> NOTE
>
> Only SDL systems v.5.0+ support this feature.

Uploading the Audio File

The first step is to make sure the audio file is available on the remote system. To do this, you use the `SDLFileManager`.

OBJECTIVE-C

```
SDLFile *audioFile = [[SDLFile alloc] initWithFileURL:<#(File location on disk)#> name:<#(Audio file's reference for usage)#> persistent:<#(True if the file is generic beyond just this session)#>];
[self.sdlManager.fileManager uploadFile:audioFile completionHandler:^(BOOL success, NSUInteger bytesAvailable, NSError * _Nullable error) {
    <#(audio file is ready if success is true)#>
}];
```

SWIFT

```
let audioFile = SDLFile(fileURL: <#File Location on disk#>, name: <#Audio file's reference for usage#>, persistent: <#True if the file is generic beyond just this session#>)
sdlManager.fileManager.upload(file: audioFile) { (success, bytesAvailable, error) in
    <#audio file is ready if success is true#>
}
```

For more information about uploading files, see the relevant guide.

Using the Audio File in an Alert

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

OBJECTIVE-C

```
SDLAlert *alert = [[SDLAlert alloc] initWithAlertText1:<#(nullable
NSString *)#> alertText2:<#(nullable NSString *)#> duration:<#(
UInt16)#>];
alert.ttsChunks = [SDLTTSChunk fileChunksWithName:<#(File's name)
#>];
[self.sdlManager sendRequest:alert];
```

SWIFT

```
let alert = SDLAlert(alertText1: <#T##String?#>, alertText2: <#T##
String?#>, duration: <#T##UInt16#>)
alert.ttsChunks = SDLTTSChunk.fileChunks(withName: <#File's name#
>)
sdlManager.send(alert)
```

Introduction

Mobile Navigation allows map partners to bring their applications into the car and display their maps and turn by turn easily for the user. This feature has a 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.

Connecting an app

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

The first difference for a navigation app is the `appHMIType` of `SDLAppHMITypeNavigation` that has to be set in the `SDLLifecycleConfiguration`. Navigation apps are also non-media apps.

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 & 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 not a general catch-all security library.

OBJECTIVE-C

```
SDLLifecycleConfiguration* lifecycleConfig = [
SDLLifecycleConfiguration defaultConfigurationWithAppName:
@"<#App Name#>" fullAppId:@"<#App Id#>"];
lifecycleConfig.appType = SDLAppHMITypeNavigation;

SDLStreamingMediaConfiguration *streamingConfig = [
SDLStreamingMediaConfiguration
secureConfigurationWithSecurityManagers:@[OEMSecurityManager.
class]];
SDLConfiguration *config = [SDLConfiguration
configurationWithLifecycle:lifecycleConfig lockScreen:[
SDLLockScreenConfiguration enabledConfiguration] logging:[
SDLLogConfiguration defaultConfiguration] streamingMedia:
streamingConfig fileManager:[SDLFileManagerConfiguration
defaultConfiguration]];
```

SWIFT

```
let lifecycleConfig = SDLLifecycleConfiguration(appName: "<#App
Name#>", fullAppId: "<#App Id#>")
lifecycleConfig.appType = .navigation

let streamingConfig = SDLStreamingMediaConfiguration(
securityManagers: [OEMSecurityManager.self])
let config = SDLConfiguration(lifecycle: lifecycleConfig, lockScreen: .
enabled(), logging: .default(), streamingMedia: streamingConfig,
fileManager: .default())
```

NOTE

When compiling, you must make sure to include all possible OEM security managers that you wish to support.

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 Menus and Keyboards guide.

Video Streaming

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 device is backgrounded or when the device is sleeping/ locked.

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 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: `autostreamingInsecureConfiguration WithInitialViewController:` or `autostreamingSecureConfigurationWithSecurityM anagers:initialViewController:`

> NOTE
>
> The View Controller you set to the `rootViewController` must be a subclass of `SDLCarWindowViewController` or have only one `supportedInterfaceOrientation`. The `SDLCarWindowViewController` 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 it's touch updates to the framerate. To disable this feature, set `SDLTouchManager.enableSyncedPanning` to `NO`.
3. `CarWindow` hard-dictates the framerate of the app. To change the framerate and other parameters, update `SDLStreamingMediaConfiguration.customVideoEncoderSettings`.

   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 `self.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 set the `rootViewController` to an off-screen view controller, i.e. you should instantiate a new `UIViewController` class and use it to set the `rootViewController`. This view controller will appear on-screen in the car, while remaining off-screen on the device. It is also possible, but not recommended, to display your on-device-screen UI to the car screen by setting the `rootViewController` to `UIApplication.sharedApplication.keyWindow.rootViewController`. However, if you mirror your device's screen, your app's UI will resize to match the head unit's screen size, thus making most of the app's UI off-screen.

> NOTE
>
> 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.
>
> 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.

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:

OBJECTIVE-C

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

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

SWIFT

```
let imageBuffer = <#Acquire Image Buffer#>

guard let streamManager = self.sdlManager.streamManager, !
streamManager.isVideoStreamingPaused else {
    return
}

if !streamManager.sendVideoData(imageBuffer) {
    print("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.

Audio Streaming

Navigation apps are allowed to stream raw audio to be played by the head unit. The audio received this way is played immediately, and the current audio source will be attenuated. The raw audio has to 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

In order to stream audio from a SDL app, we focus on the `SDLStreamingMediaManager` class. A reference to this class is available from an `SDLProxy` property `streamingMediaManager`.

Audio Stream Lifecycle

Like the lifecycle of the video stream, the lifecycle of the audio stream is maintained by the SDL library. When you recieve the `SDLAudioStreamDidStartNotification`, you can begin streaming audio.

SDLAudioStreamManager

If you do not already have raw PCM data ready at hand, the `SDLAudioStreamManager` can help. The `SDLAudioStreamManager` will help you to do on-the-fly transcoding and streaming of your files in mp3 or other formats.

OBJECTIVE-C

```
[self.sdlManager.streamManager.audioManager pushWithFileURL:
audioFileURL];
[self.sdlManager.streamManager.audioManager playNextWhenReady];
```

SWIFT

```
self.sdlManager.streamManager?.audioManager.push(withFileURL: url)
self.sdlManager.streamManager?.audioManager.playNextWhenReady()
```

IMPLEMENTING THE DELEGATE

OBJECTIVE-C

```
- (void)audioStreamManager:(SDLAudioStreamManager *)
audioManager errorDidOccurForFile:(NSURL *)fileURL error:(NSError *)
error {
}

- (void)audioStreamManager:(SDLAudioStreamManager *)
audioManager fileDidFinishPlaying:(NSURL *)fileURL successfully:(BOOL)
successfully {
    if (audioManager.queue.count != 0) {
        [audioManager playNextWhenReady];
    }
}
```

SWIFT

```
public func audioStreamManager(_ audioManager:
SDLAudioStreamManager, errorDidOccurForFile fileURL: URL, error:
Error) {

}

public func audioStreamManager(_ audioManager:
SDLAudioStreamManager, fileDidFinishPlaying fileURL: URL,
successfully: Bool) {
    if audioManager.queue.count != 0 {
        audioManager.playNextWhenReady()
    }
}
```

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.

OBJECTIVE-C

```
NSData* audioData = <#Acquire Audio Data#>;

if ([self.sdlManager.streamManager sendAudioData:audioData] == NO)
{
  NSLog(@"Could not send Audio Data");
}
```

SWIFT

```swift
let audioData = <#Acquire Audio Data#>;

guard let streamManager = self.sdlManager.streamManager,
streamManager.isAudioConnected else { return }

if streamManager.sendAudioData(audioData) == false {
    print("Could not send Audio Data")
}
```

Touch Input

Navigation applications have support for touch events, including both single and multitouch events. This includes interactions such as panning and pinch. A developer may use the included `SDLTouchManager` class, or yourself by listening to the `SDLDidReceiveTouchEventNotification` notification.

> NOTE
>
> You must have a valid and approved `appId` in order to recieve touch events.

Using SDLTouchManager

`SDLTouchManager` has multiple callbacks that will ease the implementation of touch events.

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:

OBJECTIVE-C

```
- (void)touchManager:(SDLTouchManager *)manager
didReceiveSingleTapForView:(nullable UIView *)view atPoint:(CGPoint)
point;
- (void)touchManager:(SDLTouchManager *)manager
didReceiveDoubleTapForView:(nullable UIView *)view atPoint:(CGPoint)
point;
- (void)touchManager:(SDLTouchManager *)manager
panningDidStartInView:(nullable UIView *)view atPoint:(CGPoint)point;
- (void)touchManager:(SDLTouchManager *)manager
didReceivePanningFromPoint:(CGPoint)fromPoint toPoint:(CGPoint)
toPoint;
- (void)touchManager:(SDLTouchManager *)manager
panningDidEndInView:(nullable UIView *)view atPoint:(CGPoint)point;
- (void)touchManager:(SDLTouchManager *)manager
panningCanceledAtPoint:(CGPoint)point;
- (void)touchManager:(SDLTouchManager *)manager
pinchDidStartInView:(nullable UIView *)view atCenterPoint:(CGPoint)
point;
- (void)touchManager:(SDLTouchManager *)manager
didReceivePinchAtCenterPoint:(CGPoint)point withScale:(CGFloat)scale;
- (void)touchManager:(SDLTouchManager *)manager
didReceivePinchInView:(nullable UIView *)view atCenterPoint:(CGPoint)
point withScale:(CGFloat)scale;
- (void)touchManager:(SDLTouchManager *)manager
pinchDidEndInView:(nullable UIView *)view atCenterPoint:(CGPoint)
point;
- (void)touchManager:(SDLTouchManager *)manager
pinchCanceledAtCenterPoint:(CGPoint)point;
```

SWIFT

```
    optional public func touchManager(_ manager: SDLTouchManager,
didReceiveSingleTapFor view: UIView?, at point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
didReceiveDoubleTapFor view: UIView?, at point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
panningDidStartIn view: UIView?, at point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
didReceivePanningFrom fromPoint: CGPoint, to toPoint: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
panningDidEndIn view: UIView?, at point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
panningCanceledAt point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
pinchDidStartIn view: UIView?, atCenter point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
didReceivePinchAtCenter point: CGPoint, withScale scale: CGFloat)
    optional public func touchManager(_ manager: SDLTouchManager,
didReceivePinchIn view: UIView?, atCenter point: CGPoint, withScale
scale: CGFloat)
    optional public func touchManager(_ manager: SDLTouchManager,
pinchDidEndIn view: UIView?, atCenter point: CGPoint)
    optional public func touchManager(_ manager: SDLTouchManager,
pinchCanceledAtCenter point: CGPoint)
```

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 apps want to have access to the raw touch data, the `SDLDidReceiveTouchEventNotification` notification can be evaluated. This callback will be fired for every touch of the user and contains 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

OBJECTIVE-C

```
[[NSNotificationCenter defaultCenter] addObserver:self selector:
@selector(touchEventAvailable:) name:
SDLDidReceiveTouchEventNotification object:nil];

- (void)touchEventAvailable:(SDLRPCNotificationNotification *)
notification {
    if (![notification.notification isKindOfClass:SDLOnTouchEvent.class]) {
      return;
    }
    SDLOnTouchEvent *touchEvent = (SDLOnTouchEvent *)notification.
notification;

    // Grab something like type
    SDLTouchType* type = touchEvent.type;
}
```

SWIFT

```
// To Register
NotificationCenter.default.addObserver(self, selector: #selector(
touchEventAvailable(_:)), name: .SDLDidReceiveTouchEvent, object: nil)

// On Receive
@objc private func touchEventAvailable(_ notification:
SDLRPCNotificationNotification) {
    guard let touchEvent = notification.notification as? SDLOnTouchEvent
 else {
        print("Error retrieving onTouchEvent object")
        return
    }

    // Grab something like type
    let type = touchEvent.type
}
```

Supporting Haptic Input

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 areas 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 rectangle, the center of that area will be "touched".

> NOTE
>
> Currently, there are no RPCs for knowing which rect is highlighted, so your UI will have to remain static, without scrolling.

You will also need to implement touch input support (Mobile Navigation/Touch Input) in order to receive touches of the rects. You must support the automatic focusable item manager in order to receive a touched view back in the `SDLTouchManagerDelegate` in addition to the `CGPoint`.

Automatic Focusable Rects

SDL has support for automatically detecting focusable rects 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 rects to be sent.

NOTE

This is only supported on iOS 9 devices and above. If you want to support this on iOS 8, see "Manual Focusable Rects" below.

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 device is on iOS 9+ and the window is set, the focusable item locator will start running. Whenever your app updates, you will need to send a notification:

OBJECTIVE-C

```
[[NSNotificationCenter defaultCenter] postNotificationName:
SDLDidUpdateProjectionView object:nil];
```

SWIFT

```
NotificationCenter.default.post(name: SDLDidUpdateProjectionView,
object: nil)
```

NOTE

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".

Manual Focusable Rects

If you need to supplement the automatic focusable item locator, or do all of the location yourself (e.g. devices lower than iOS 9, or 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`.

OBJECTIVE-C

```
SDLRectange *viewRect = [[SDLRectangle alloc] initWithCGRect:view.
bounds];
SDLHapticRect *hapticRect = [[SDLHapticRect alloc] initWithId:1 rect:
viewRect];
SDLSendHapticData *hapticData = [[SDLSendHapticData alloc]
initWithHapticRectData:@[hapticRect]];

[self.sdlManager.sendRequest:hapticData];
```

SWIFT

```swift
guard let viewRect = SDLRectange(cgRect: view.bounds) else { return }
let hapticRect = SDLHapticRect(id: 1, rect: viewRect)
let hapticData = SDLSendHapticData(hapticRectData: [hapticRect])

self.sdlManager.send(hapticData)
```

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.

To display a Turn by Turn direction, a combination of the `SDLShowConstantTBT` and `SDLAlertManeuver` RPCs must be used. The `SDLShowConstantTBT` RPC involves the data that will be shown on the head unit. The main properties of this object to set are `navigationText1`, `navigationText2`, and `turnIcon`. A best practice for navigation applications is to use the `navigationText1` as the direction to give (Turn Right) and `navigationText2` to provide the distance to that direction (3 mi). When an `SDLAlertManeuver` is sent, you may also include accompanying text that you would like the head unit to speak when an direction is displayed on screen (e.g. In 3 miles turn right.).

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

Sending a Maneuver

OBJECTIVE-C

```
// Create SDLImage object for turnIcon.
SDLImage* turnIcon = <#Create SDLImage#>;

SDLShowConstantTBT* turnByTurn = [[SDLShowConstantTBT alloc] init
];
turnByTurn.navigationText1 = @"Turn Right";
turnByTurn.navigationText2 = @"3 mi";
turnByTurn.turnIcon = turnIcon;

__weak typeof(self) weakSelf = self;
[self.sdlManager sendRequest:turnByTurn withResponseHandler:^(
SDLRPCRequest *request, SDLRPCResponse *response, NSError *error)
{
    if (![response.resultCode isEqualToEnum:SDLResultSuccess]) {
        NSLog(@"Error sending TBT");
        return;
    }

    typeof(weakSelf) strongSelf = weakSelf;
    SDLAlertManeuver* alertManeuver = [[SDLAlertManeuver alloc]
initWithTTS:@"In 3 miles turn right" softButtons:nil];
    [strongSelf.sdlManager sendRequest:alertManeuver
withResponseHandler:^(SDLRPCRequest *request, SDLRPCResponse *
response, NSError *error) {
        if (![response.resultCode isEqualToEnum:SDLResultSuccess]) {
            NSLog(@"Error sending AlertManeuver.");
            return;
        }
    }];
}];
```

SWIFT

```
// Create SDLImage object for turnIcon.
let turnIcon = <#Create SDLImage#>

let turnByTurn = SDLShowConstantTBT()
turnByTurn.navigationText1 = "Turn Right"
turnByTurn.navigationText2 = "3 mi"
turnByTurn.turnIcon = turnIcon

sdlManager.send(request: turnByTurn) { [weak self] (request, response,
 error) in
    if response?.resultCode.isEqual(to: .success) == false {
        print("Error sending TBT.")
        return
    }

    let alertManeuver = SDLAlertManeuver(tts: "In 3 miles turn right",
softButtons: nil)
    self.sdlManager.send(alertManeuver) { (request, response, error) in
        if response?.resultCode.isEqual(to: .success) == false {
            print("Error sending AlertManeuver.")
            return
        }
    }
}
```

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

Clearing the Maneuver

To clear a navigation direction from the screen, we send an `SDLShowConstantTBT` with the `maneuverComplete` property as `YES`. This specific RPC does not require an accompanying `SDLAlertManeuver`.

OBJECTIVE-C

```
SDLShowConstantTBT* turnByTurn = [[SDLShowConstantTBT alloc] init
];
turnByTurn.maneuverComplete = @(YES);

[self.sdlManager sendRequest:turnByTurn withResponseHandler:^(
SDLRPCRequest *request, SDLRPCResponse *response, NSError *error)
{
    if (![response.resultCode isEqualToEnum:SDLResultSuccess]) {
        NSLog(@"Error sending TBT.");
        return;
    }

    // Successfully cleared
}];
```

SWIFT

```
let turnByTurn = SDLShowConstantTBT()
turnByTurn.maneuverComplete = true

sdlManager.send(request: turnByTurn) { (request, response, error) in
    if response?.resultCode.isEqual(to: .success) == false {
        print("Error sending TBT.")
        return
    }
}
```

Configuring SDL Logging

SDL iOS v5.0 includes a powerful new built-in logging framework designed to make debugging 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:

OBJECTIVE-C

```
SDLConfiguration* configuration = [SDLConfiguration
configurationWithLifecycle:lifecycleConfiguration lockScreen:[
SDLLockScreenConfiguration enabledConfiguration] logging:[
SDLLogConfiguration debugConfiguration] fileManager:[
SDLFileManagerConfiguration defaultConfiguration]];
```

SWIFT

```
let configuration = SDLConfiguration(lifecycle: lifecycleConfiguration,
lockScreen: .enabled(), logging: .debug(), fileManager: .default())
```

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, in both default and debug configurations, only the Apple System Logger target (iOS 9 and below) or OSLog (iOS 10+) will be enabled.

APPLE SYSTEM LOG TARGET

The Apple System Logger target, `SDLLogTargetAppleSystemLogger`, is the default log target for both default and debug configurations on devices running iOS 9 or older. This 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 devices running iOS 10 or newer. 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 MacOS Sierra's 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.

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:

OBJECTIVE-C

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

SWIFT

```
let _ = logConfig.targets.insert(SDLLogTargetFile())
```

Modules

A module is a set of files packaged together. Create modules using the `SDLLog FileModule` 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:

OBJECTIVE-C

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

SWIFT

```
logConfig.modules.insert(SDLLogFileModule(name: "Test", files: ["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.

OBJECTIVE-C

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

SWIFT

```
let filter = SDLLogFilter(disallowingString: "Test", caseSensitive: false)
```

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 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
```

LOGGING IN SWIFT

After the submodule has been installed, 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")
```

SDL Relay

> NOTE
>
> A better method than using the iOS Relay app is to use Xcode 9 / iOS 11 wireless debugging. This allows you to use the IAP connection instead of the TCP connection and can be more reliable. The Relay app is still useful if your phone is not on iOS 11.

The SmartDeviceLink (SDL) iOS Relay app is a debugging tool for developers building iOS applications that communicate with a vehicle head unit requiring a USB cable. Testing is done over a TCP/IP connection. During testing developers can easily see logs of in-going/out-going remote procedure calls (RPCs) in Xcode's debug console, thus making debugging easier and faster.

NOTE

If you are using a head unit or TDK, and are using the Relay app for debugging, the IP address and port number should be set to the same IP address and port number as the app. This information appears in the Relay app once the server is turned on in the app. Also be sure that the device is on the same network as your app.

Necessary Tools

In order to use the Relay app, you must have the following tools:

1. A SDL Core enabled vehicle head unit or a Test Development Kit.
2. An iOS device with the SDL iOS Relay app installed.
3. A USB cable to connect the iOS device to the head unit or TDK.
4. Xcode with the app being tested running in Xcode's iOS Simulator app.

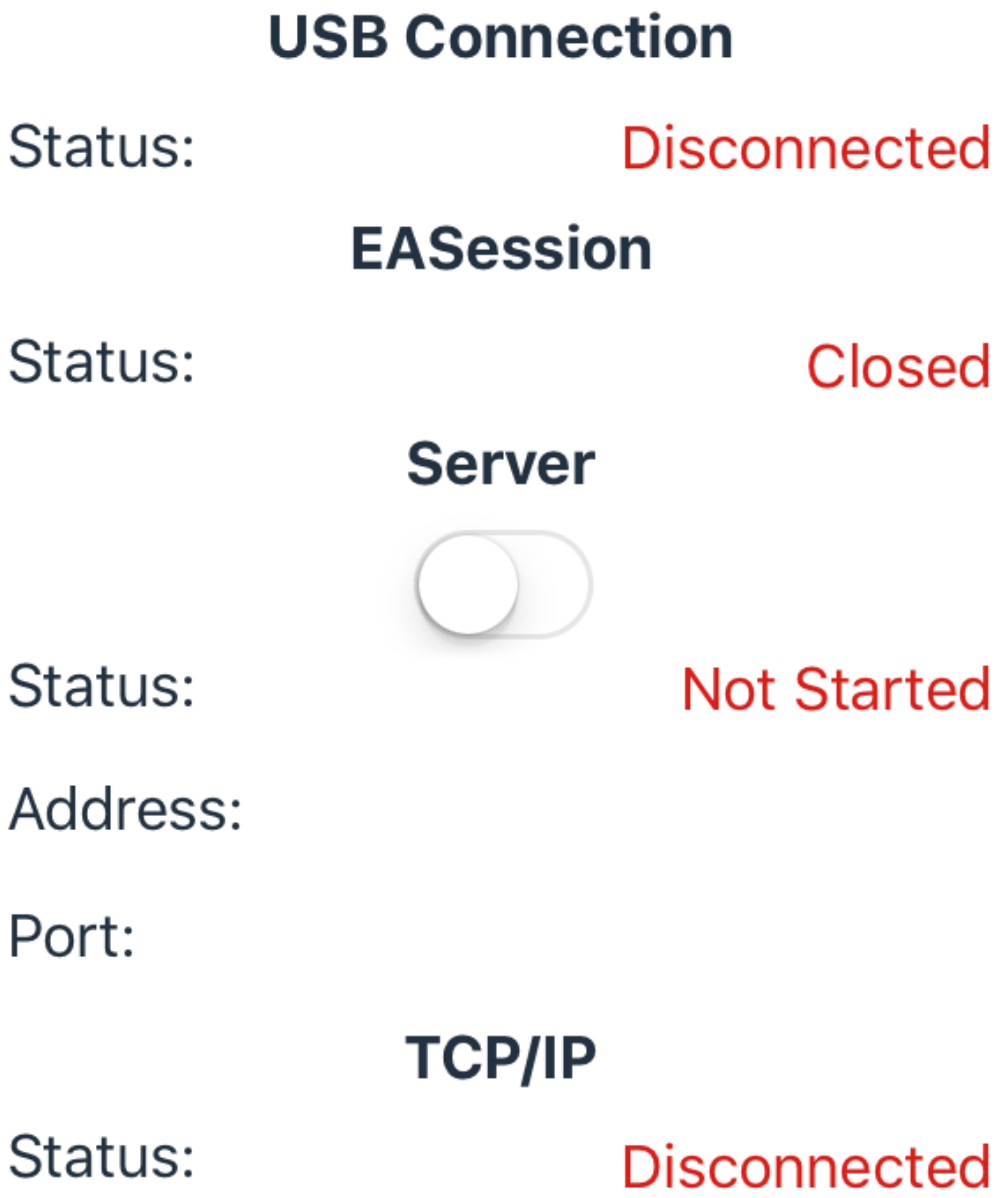
Examples

Example: Connecting the RPC Builder iOS App

This example shows you how to use the RPC Builder app in conjunction with the Relay app. For a tutorial on how to connect a custom app, please see the example below.

1. Download the RPC Builder app. The RPC Builder app is a free tool designed to help developers understand how RPCs work.
2. Download the Relay app and install it on an iOS device.

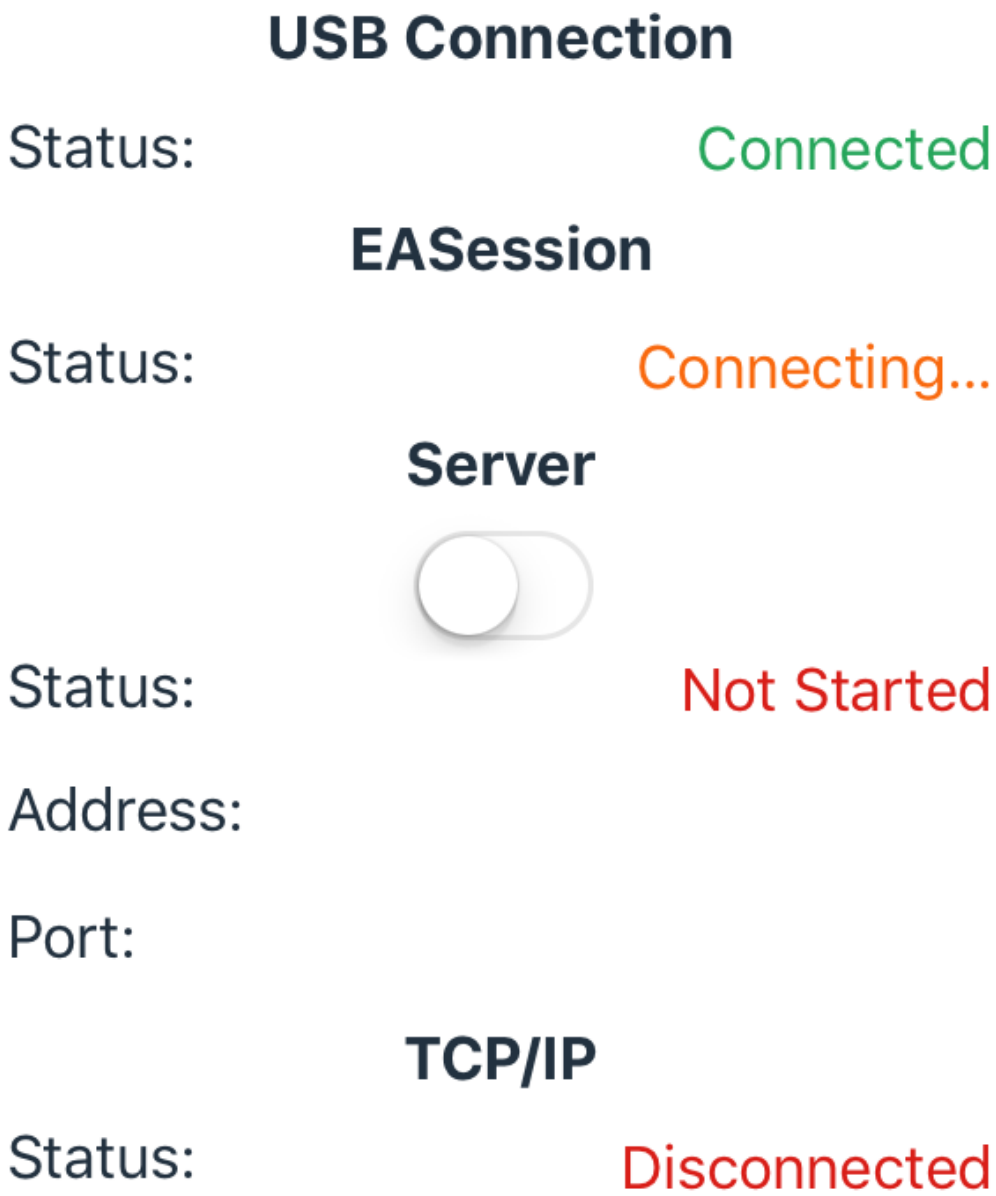
3. Launch the Relay app on an iOS device. If the Relay app is not connected to any hardware running SDL Core via USB, the app's screen will not show any active connections.



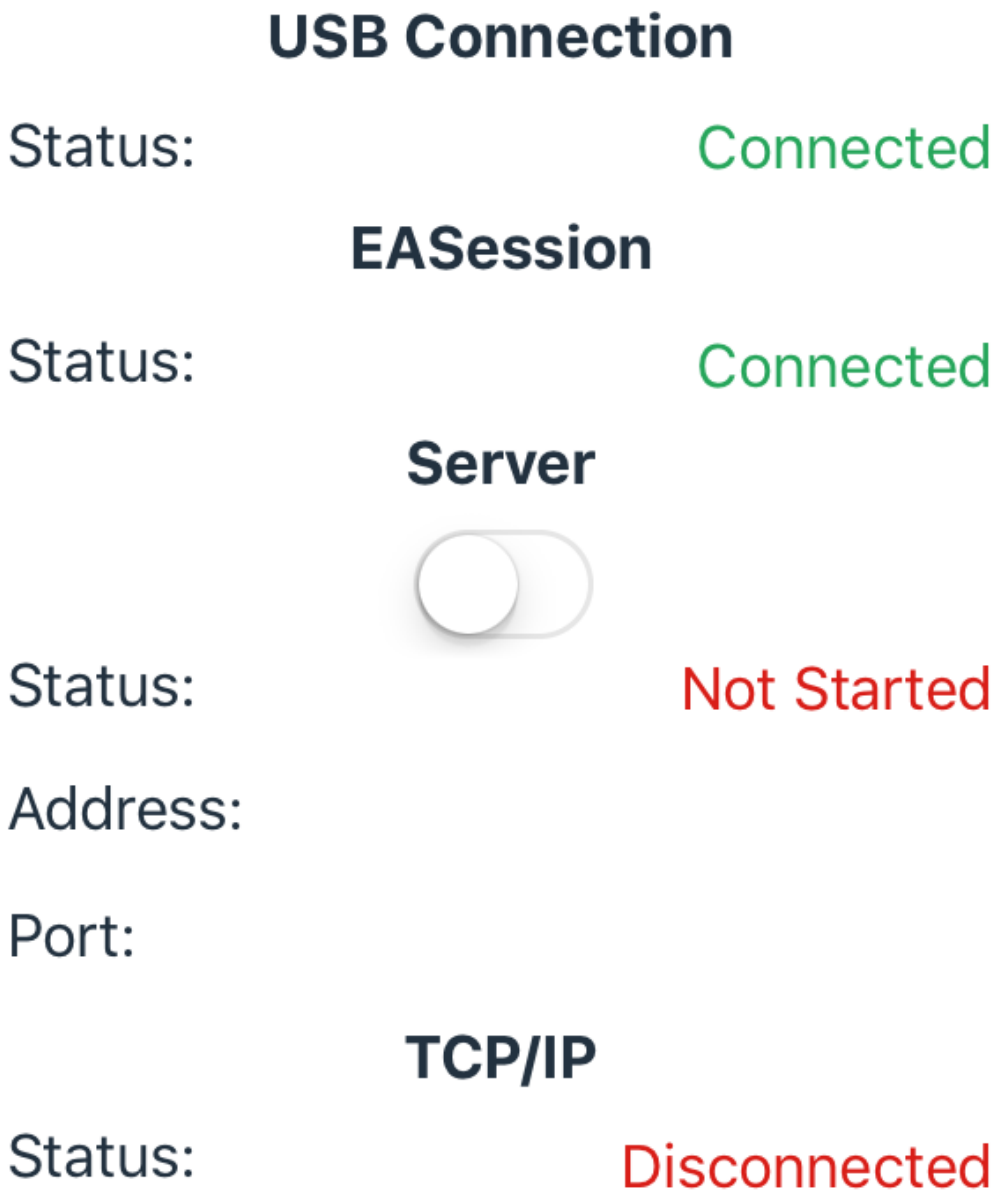
Initial app startup. This state is visible when the app is not connected to hardware running SDL Core via USB.

4. Connect the iOS device to the SDL Core using a USB cable.
5. When the iOS device is connected to the SDL Core, the status under **USB Connection** should change from *Disconnected* to *Connected*. Wait

for the the status of the **EASession** to change to *Connected*. The EASession is a communication channel between the Relay app and SDL Core. If **EASession** is not *Connected* within a few seconds pull and reconnect the USB cord to the SDL Core.

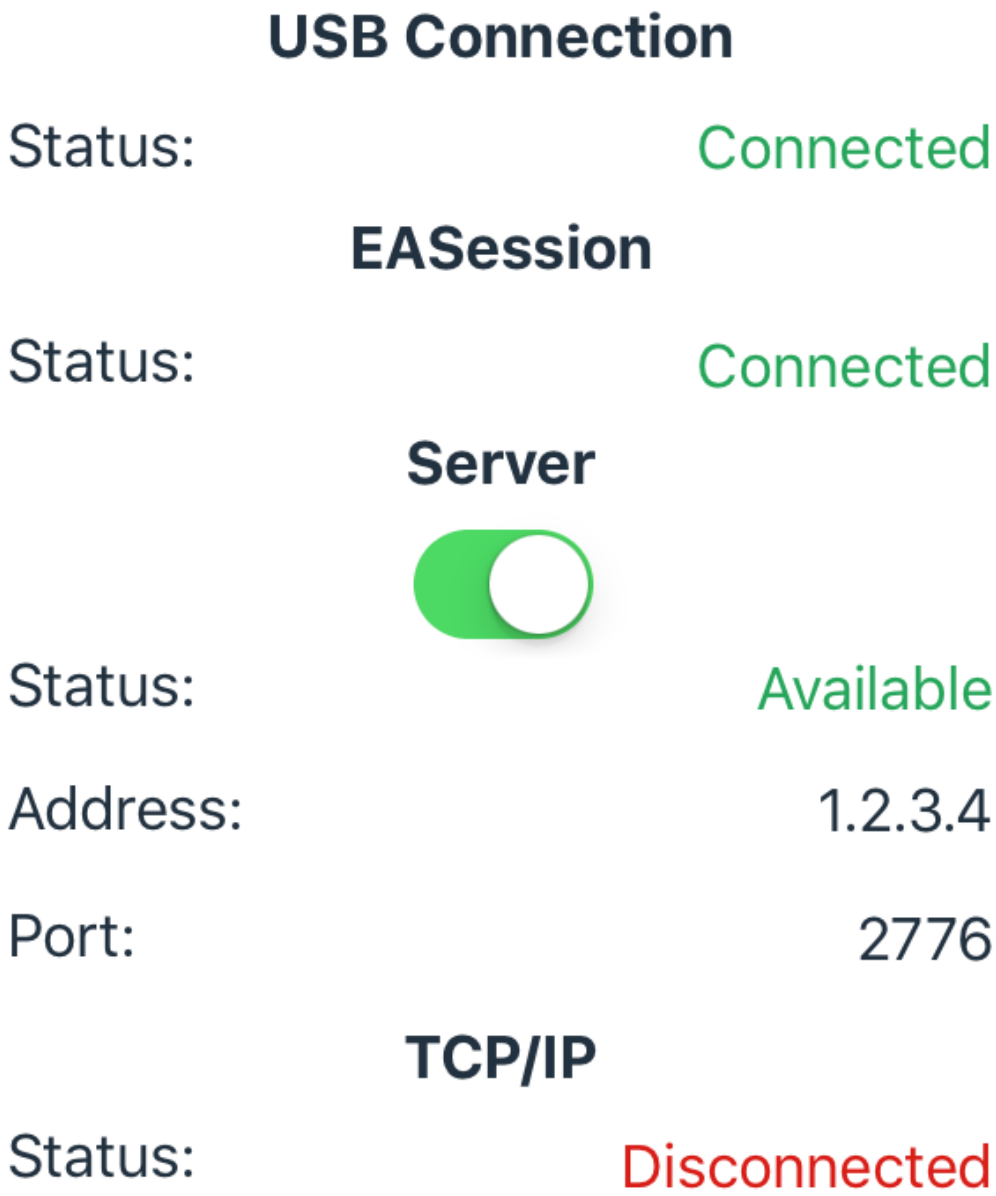


The Relay App is initially connected via USB, but the connection is still in progress.



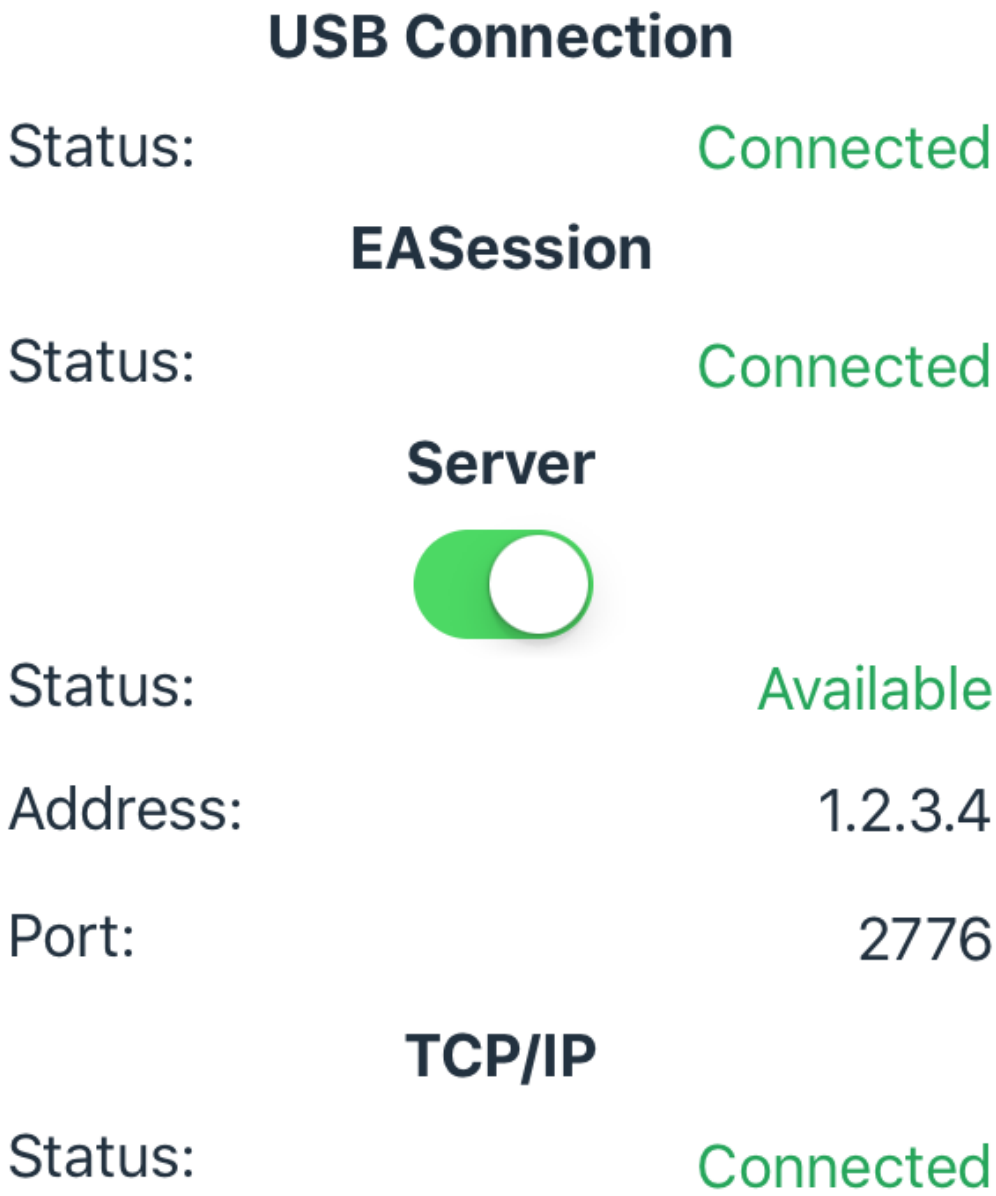
The Relay App is fully connected via USB, and ready for server start.

6. Once the USB Connection and EASession are both set to *Connected*, the app is fully connected and ready for server start. Toggle the switch under **Server** to on. When the status of the server changes to *Available*, the IP address and port number of the wifi network the Relay app is connected to will appear under **Server**.



Server is now started and awaiting connection.

7. Open the RPC Builder app in Xcode and click on the *run* button to launch the app in Xcode's iOS Simulator. Enter the IP address and port number from the Relay app into the RPC Builder app and click on *Next*. On the next page of the RPC Builder app, click on *Send*.
8. Once the RPC Builder app is running on the Simulator, the status of **SDL** in the Relay app should change to *Connected*.



Application is correctly connected to Relay, and messages can now be sent and received.

9. The RPC Builder app is now connected to Relay, and messages can be sent and received. Debug logs will appear in Xcode's debug area.

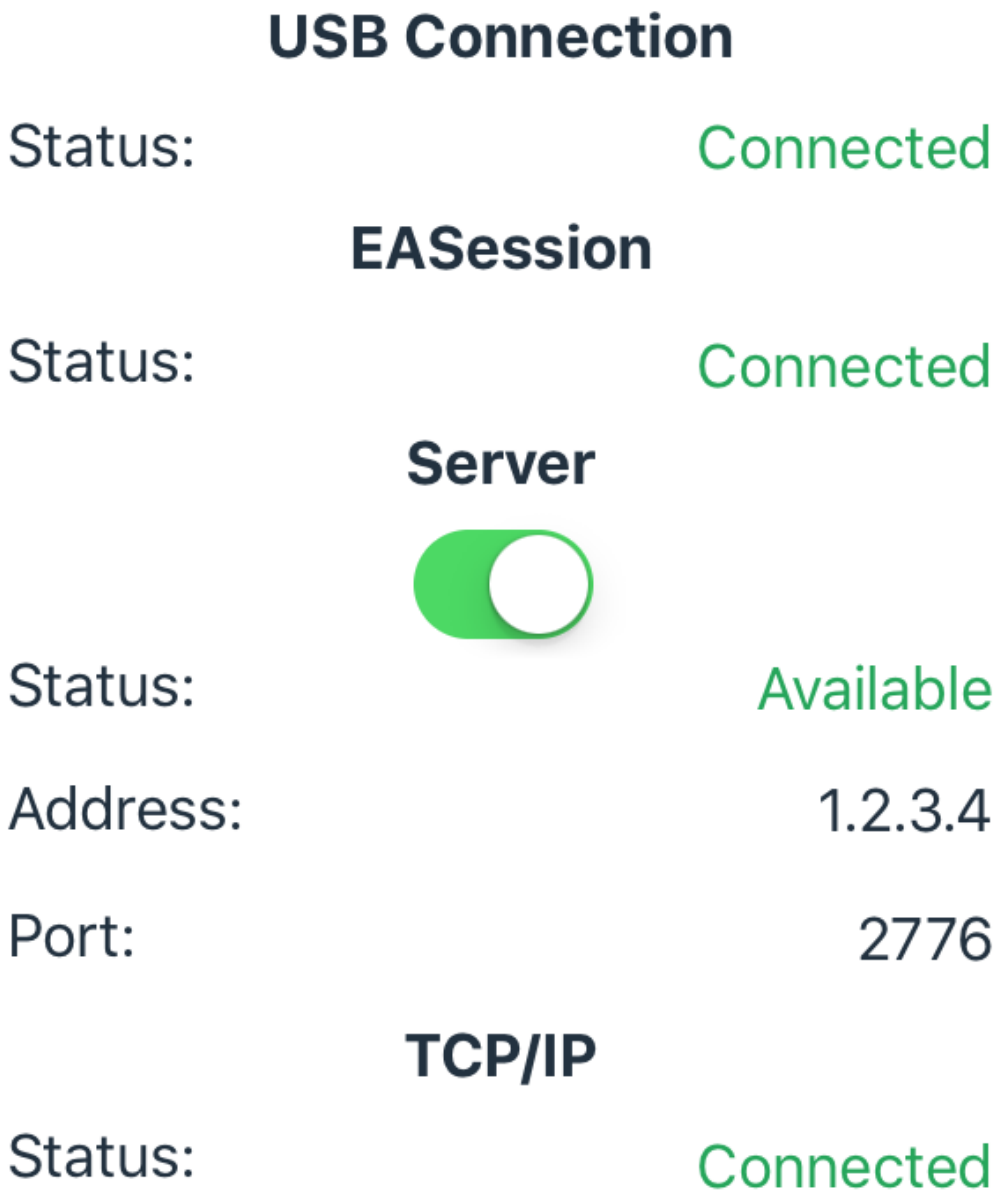
Example: Connecting Your Custom App

This example shows you how to connect a custom app with the Relay app.

1. First, follow steps 2 through 7 in the example above called *Connecting the RPC Builder iOS App*.
2. It is very important to make sure that the Relay app and the app you are testing are connected to the same wifi network. Make sure to set the proxy's TCP/IP initializer with the same IP address and port number used by the Relay app. To do this, set the proxy builder's TCP/IP initializer in the app being tested.

```
SDLProxy* proxy = [SDLProxyFactory buildSDLProxyWithListener:
sdlProxyListenerDelegate
    tcpIPAddress:@"1.2.3.4"
    port:@"2776"];
```

3. Start the app being tested on Xcode's simulator.
4. Once the app is running on the simulator, the status of **SDL** in the Relay app should change to *Connected*.



Application is correctly connected to Relay, and messages can now be sent and received.

5. The app is now connected to Relay, and messages can be sent and received. Debug logs will appear in Xcode's debug area.

NOTE

The Relay app should always be connected to the SDL Core before starting your app, otherwise setup will not work.

Need Help?

If you need general assistance, or have other questions, you can sign up for the SDL Slack and chat with other developers and the maintainers of the project.

Found a Bug?

If you see a bug, feel free to post an issue.

Want to Help?

If you want to help add more features, please file a pull request.

RPC Builder

Introduction

The SmartDeviceLink (SDL) RPC Builder app is a free iOS app designed to help developers understand the SDL interface and how remote procedure calls (RPCs) work. Use the app to test sending and receiving RPCs without writing any code.

NOTE

In order for the RPC Builder app to work correctly, all commands must be executed in proper sequence. For example, when building a custom menu, a `performInteraction` request will only be successful if sent after a `createInteractionChoiceSet` request. To find more information about how to properly set up a sequence of commands, please reference the SDL App Developer Documentation.

Getting Started

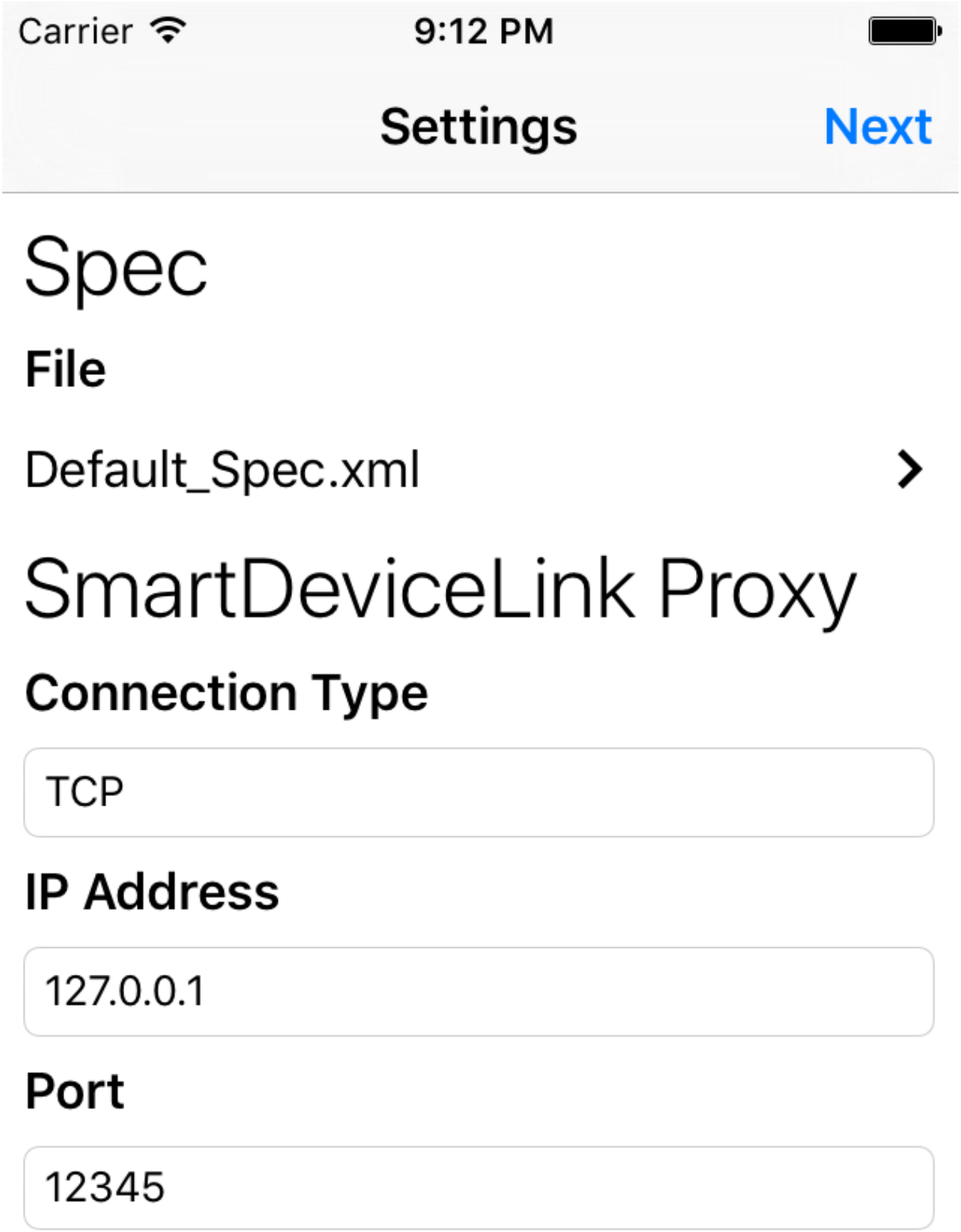
In order to begin using RPC Builder, the SDL iOS library must be added to the project. There is already support for CocoaPods in this project, so to install the library, simply navigate to the RPC Builder folder in a terminal and then install:

```
cd RPC\ Builder/
pod install
```

Once the SDL iOS library has been installed, the RPC Builder app can be deployed on an iOS device.

RPC Builder Interface

Settings Page

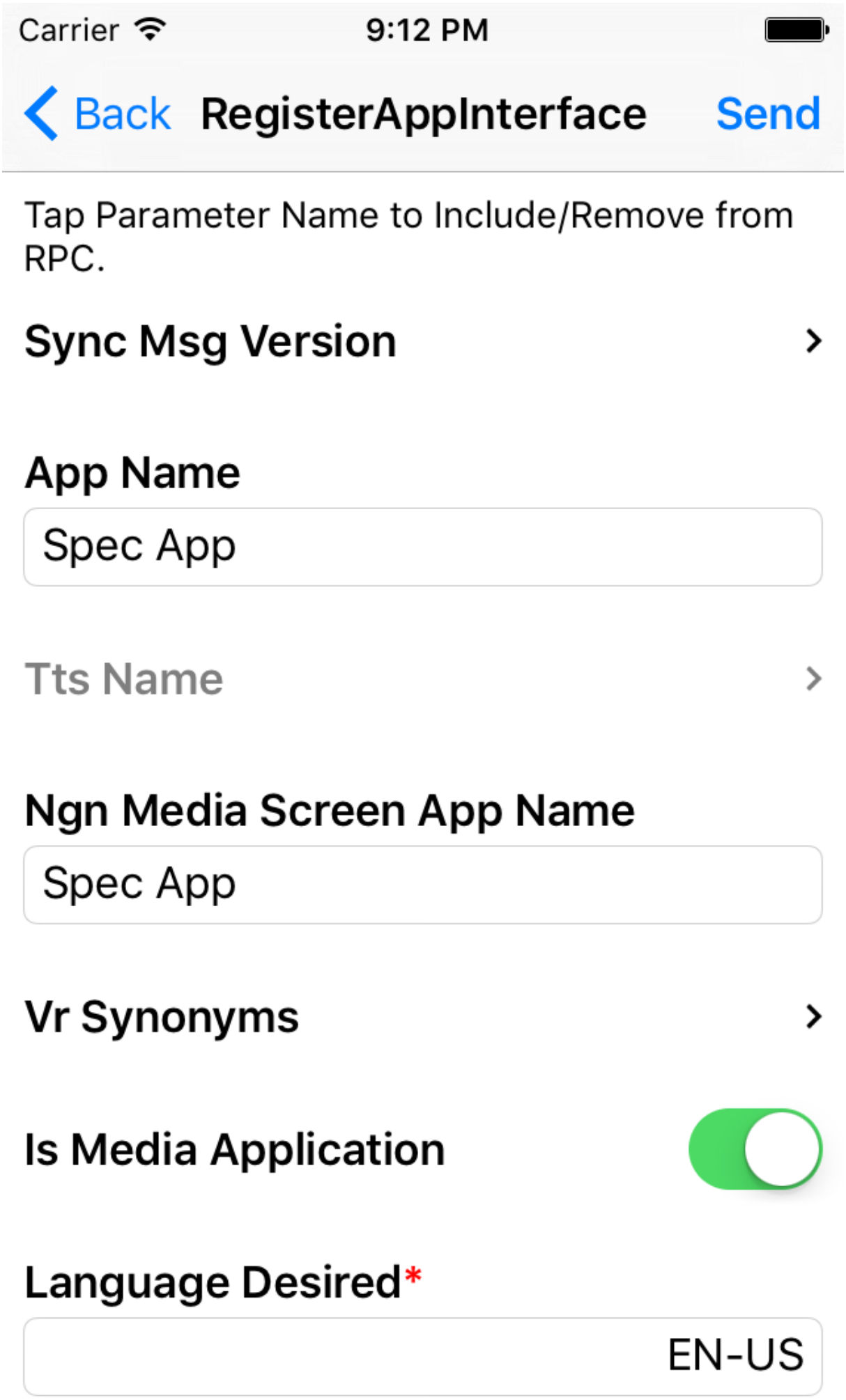


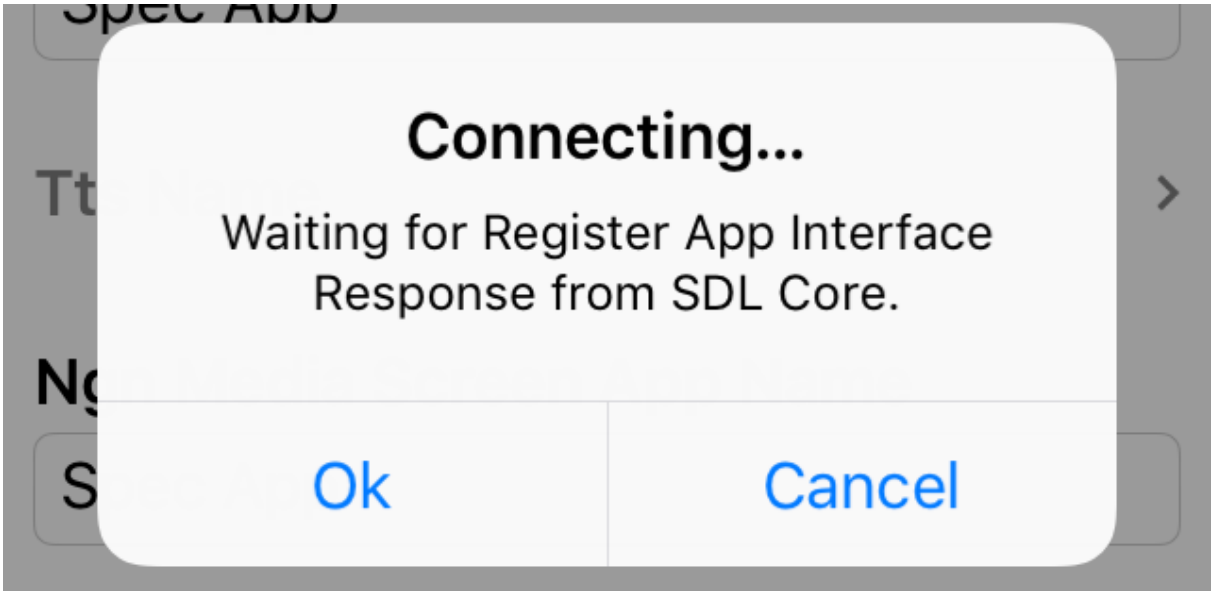
On the settings page, select a RPC spec file. The default *Mobile_API.xml* file will generate all possible RPCs available for the app. To use a custom RPC spec file,

add a new file via iTunes file sharing to the `SpecXMLs` directory. The file can also be added via a remote URL.

Also on the settings page, set the transport layer to TCP/IP or iAP. For more information on which type of connection to use, please view the SDL iOS Guide.

Once the spec file and transport layer have been set, click on *Next*. On the next page, send the `RegisterAppInterface` (RAI) RPC, a request that registers the app with SDL Core. Simply click on *Send* to use the default RAI settings. If the properties on the RAI screen are modified, they will be cached for subsequent launches.

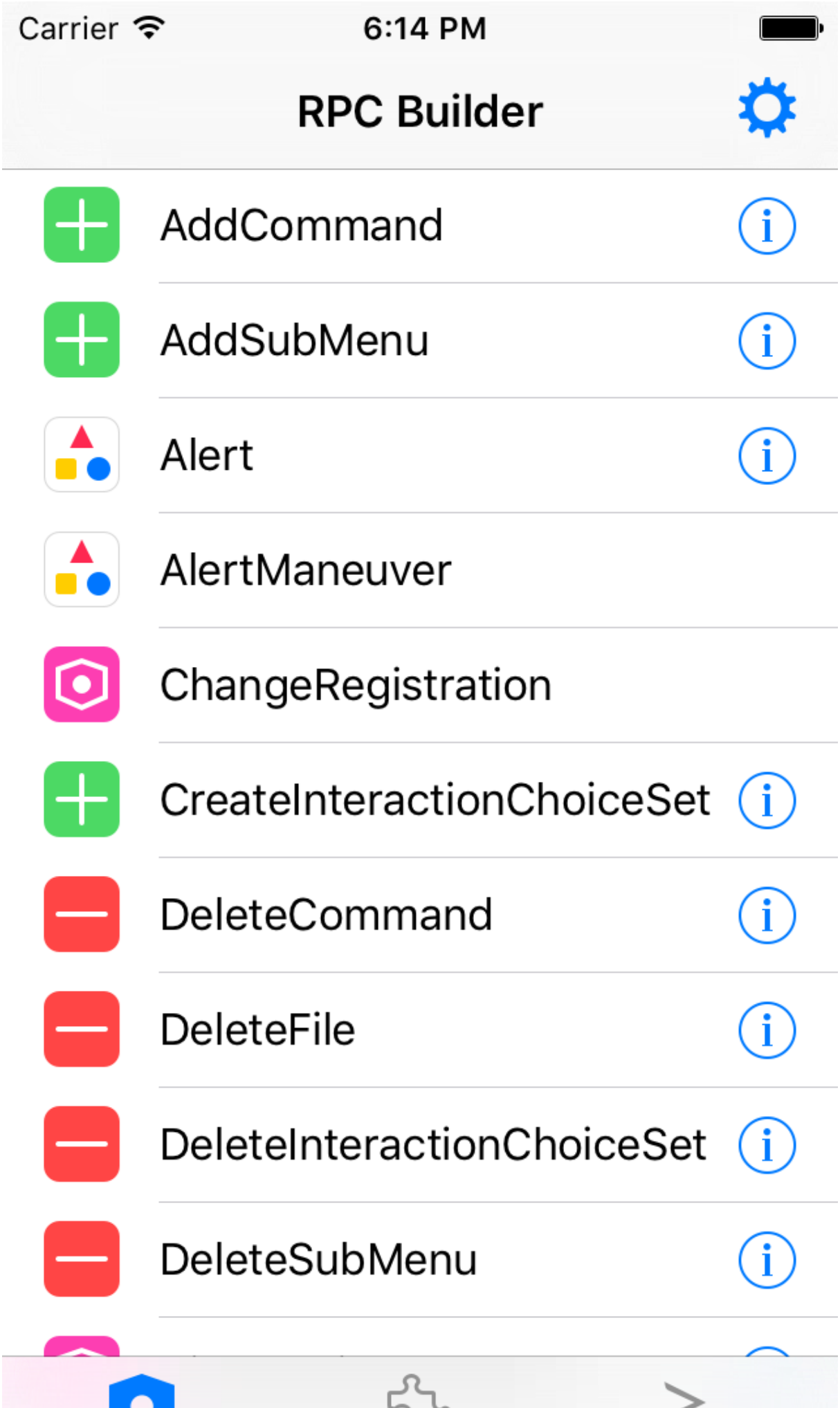

Carrier
9:12 PM
Back
RegisterAppInterface
Send
Tap Parameter Name to Include/Remove from RPC.
Sync Msg Version
App Name
Spec App
Tts Name
Ngn Media Screen App Name
Spec App
Vr Synonyms
Is Media Application
Language Desired*
EN-US



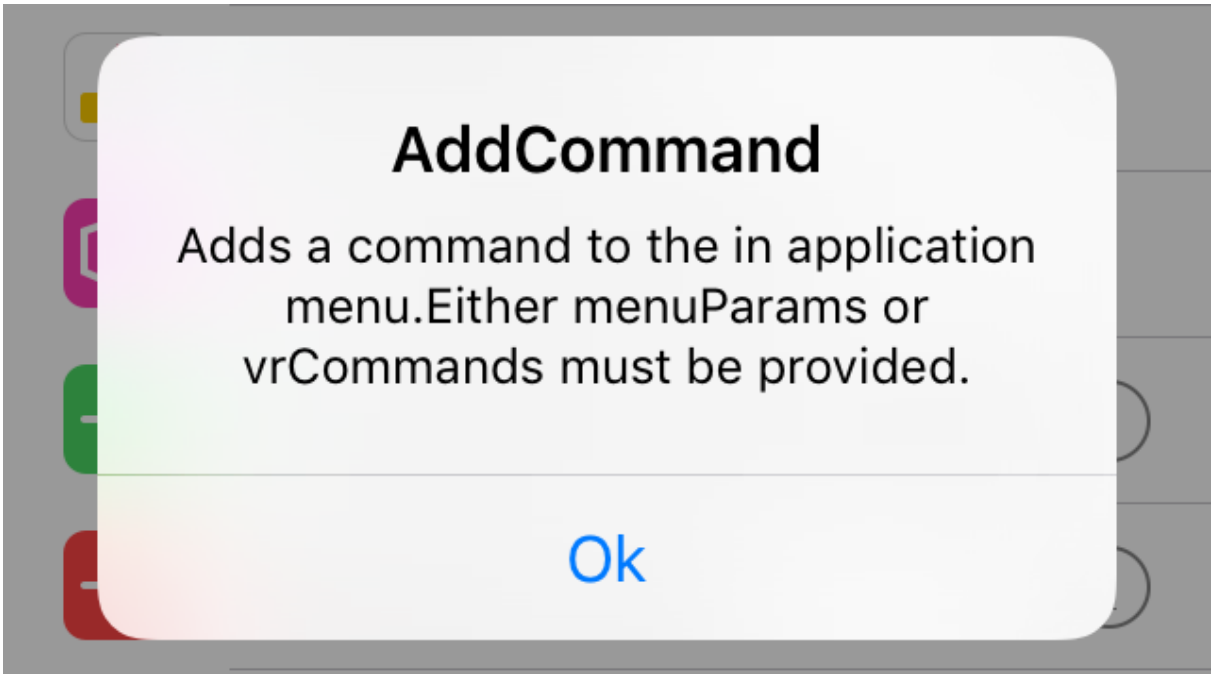
NOTE

Once *Send* is pressed, the app will only proceed once it has successfully connected with SDL Core and received a RAI response.

Main RPCs Table


Carrier
6:14 PM
RPC Builder
AddCommand
AddSubMenu
Alert
AlertManeuver
ChangeRegistration
CreateInteractionChoiceSet
DeleteCommand
DeleteFile
DeleteInteractionChoiceSet
DeleteSubMenu

The main RPC table is created at runtime by the app from a spec XML file. If there is additional information provided about the RPC call, an information button will appear next to the RPC name in the table. Click on the information button to learn more about the RPC call.

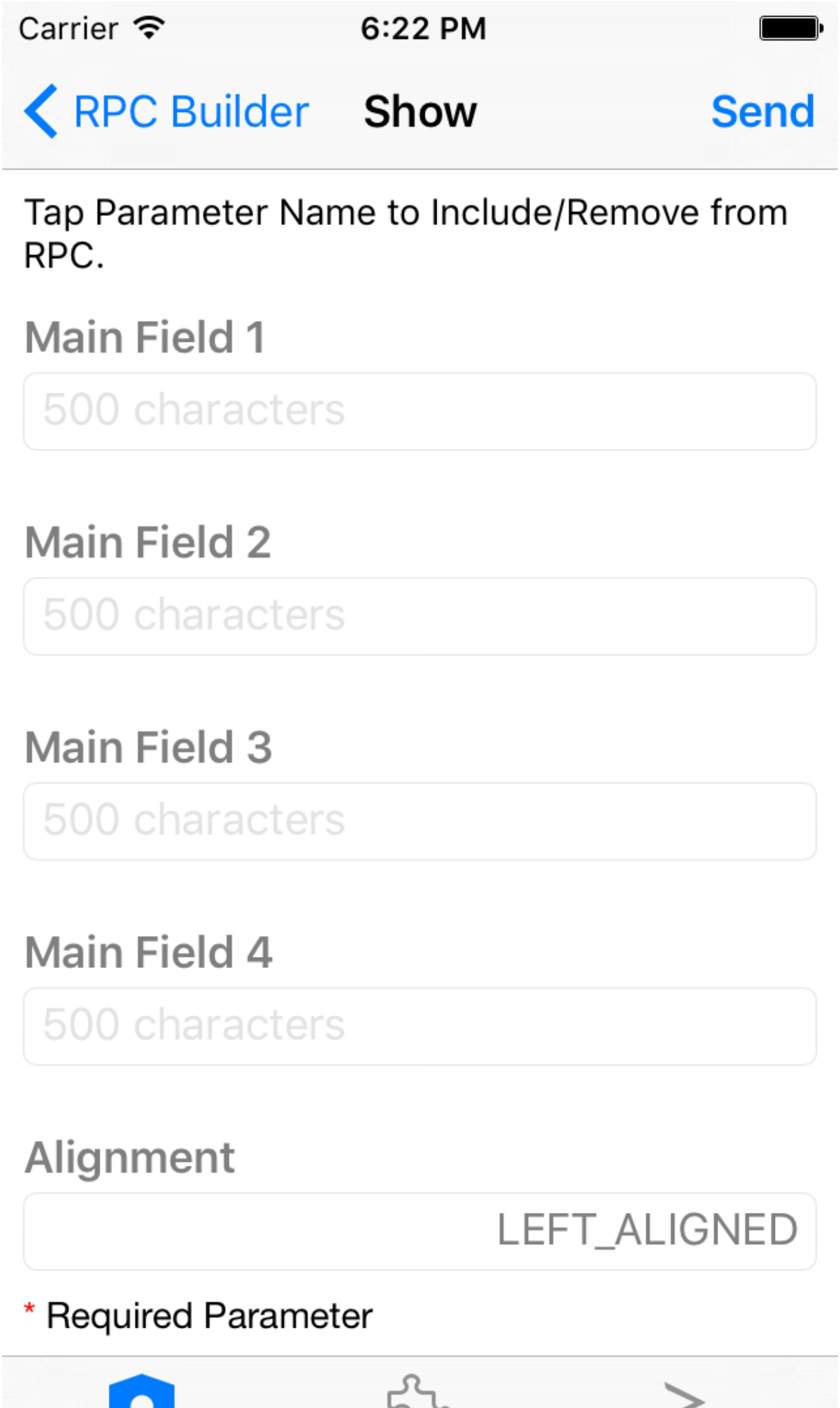


Send a RPC

To send an RPC to the SDL Core select the RPC from the table, fill out the RPC parameters and click *Send*.

PARAMETER INFORMATION

After selecting an RPC from the table, a view will appear with all possible parameters for this RPC. To find out more information about an argument, tap and hold the argument name to reveal the information.


Carrier
6:22 PM
RPC Builder
Show
Send
Tap Parameter Name to Include/Remove from RPC.
Main Field 1
500 characters
Main Field 2
500 characters
Main Field 3
500 characters
Main Field 4
500 characters
Alignment
LEFT_ALIGNED
* Required Parameter

REQUIRED PARAMETERS

Required data will have a red asterisk next to the argument name.

STRUCT OR ARRAY PARAMETERS

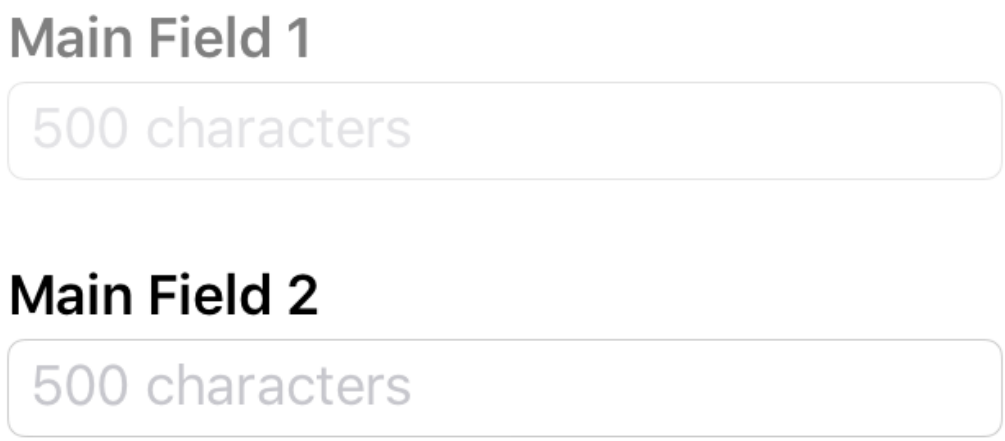
If a parameter is a struct or array, an arrow will appear to the right of the parameter name. More parameter information for the array or struct can be entered by clicking on the parameter table cell. A new view will appear where more information about the parameter can be entered.

PARAMETER DATA

There are three different ways to send an RPC argument.

1. Send with data.
    - To send an argument with data just add the information next to the arguments name.
2. Send without data
    - To send an argument with an empty string, leave the field next to the argument name empty
3. Don't send the argument
    - To disable the argument from being included in the RPC, tap once on the argument's name. The argument will be grayed out and not included in the request. In the picture below `mainField1` will not be included in the RPC Request, but `mainField2` will be included with an

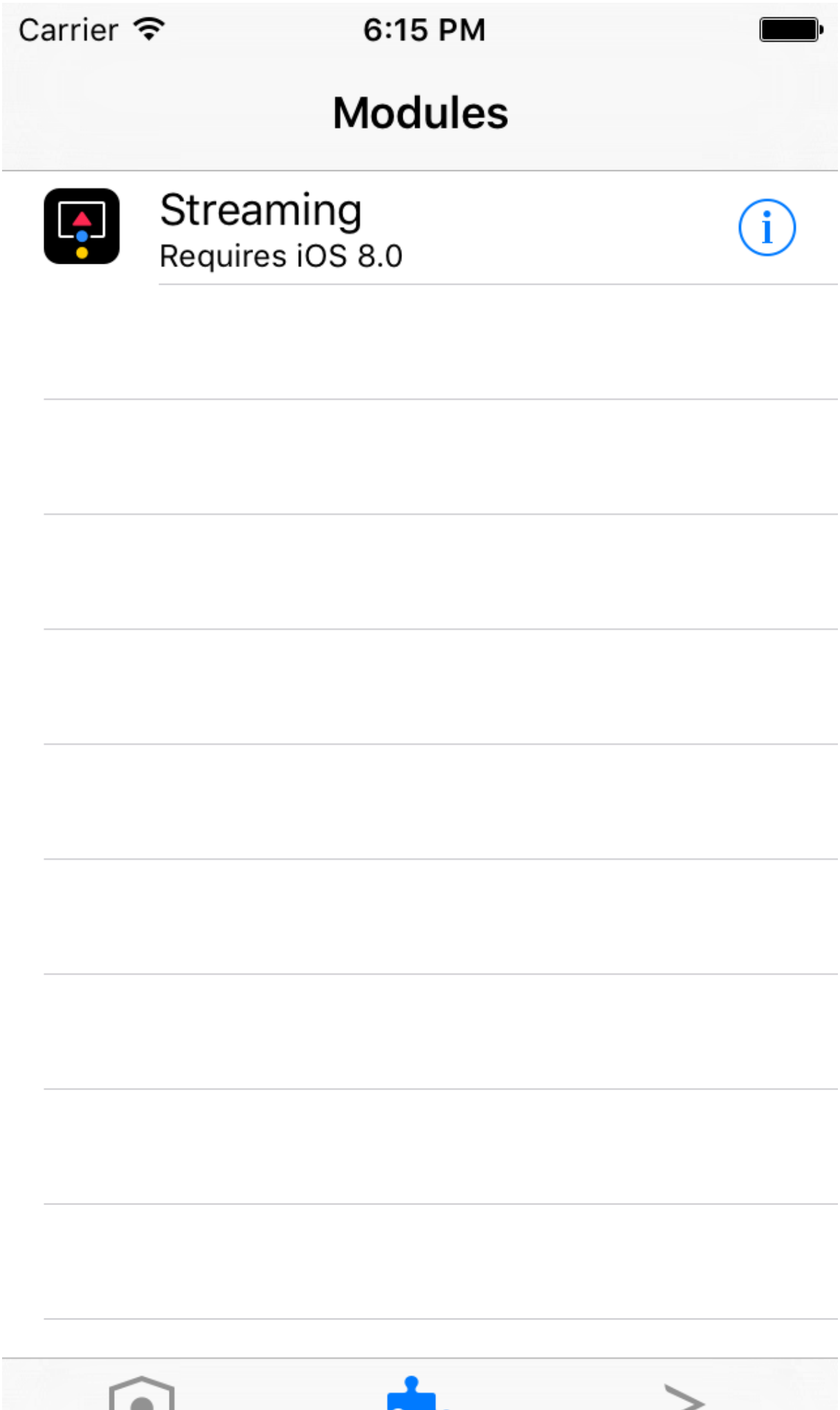
empty string.



Modules

The purpose of modules is to allow developers to create more advanced testing scenarios. A module can contain multiple RPCs. It can also define capabilities

not provided in the RPC spec file.


Carrier
6:15 PM
Modules
Streaming
Requires iOS 8.0

BUILDING NEW MODULES

There are a few requirements for building Modules:

1. All Modules must be subclasses of `RBModuleViewController`, and all class functions labeled as **Required** must be overridden.
    - These properties will allow other developers to easily understand what the Module will be testing and will also include the iOS version required in order to use Module.
    - Any Module with an iOS version other than 6 as the requirement will be listed.
    - Although other class functions such as `moduleImageName`/ `moduleImage` are optional, it is encouraged to add these functions.
2. All Modules must use the provided `SDLProxy`, `SDLManager`, and `RBSettingsManager` that are provided to subclasses of `RBModuleViewController`.
3. All Modules must be added to the `Modules.storyboard` storyboard in order to correctly load.
    - When designing your view controller, use 8px for the vertical and horizontal displacement between views to create a consistent user experience.
4. All Modules must not interact with any other Module.
5. All Modules must be added to `RBModuleViewController`'s class function `moduleClassNames`. The new Module should be added to this list in alphabetical order. For an example of how to add this see below:

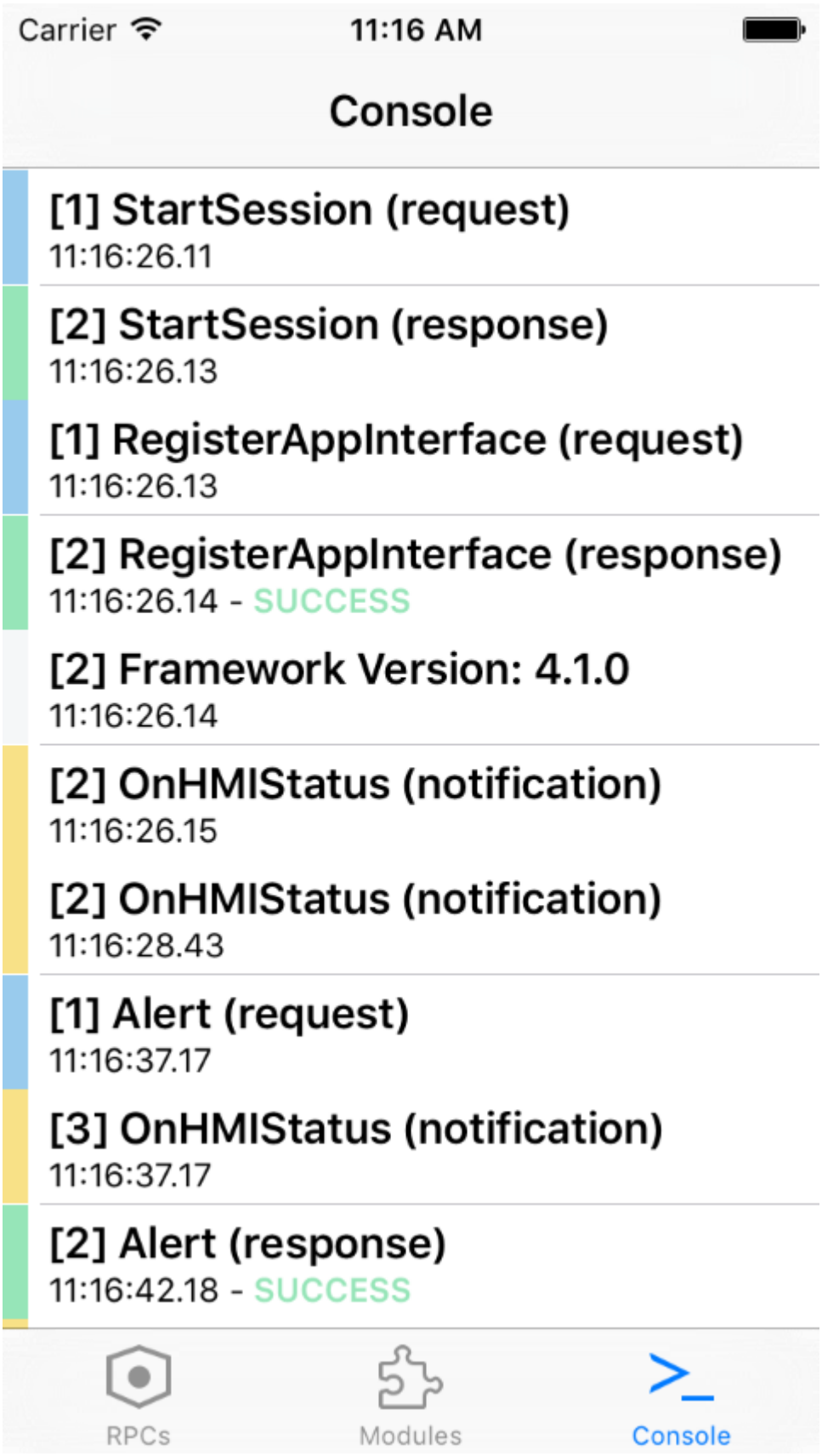
```
+ (NSArray*)moduleClassNames {
      if (!moduleClassNames) {
            moduleClassNames = @[
            [RBStreamingModuleViewController classString],  //
Streaming
            [RBNewModuleViewController classString]  // Module
Name
            ];
      }
      return moduleClassNames;
}
```

DEFAULT MODULES

1. Streaming
    - Allows for testing of video and audio streaming of camera / video files as well as audio files respectively.
2. Audio Capture
    - Allows for saving of audio data via `AudioPassThru` RPCs. Properties of this RPC can be modified to test the robustness of the RPC. This audio data may be retrieved via iTunes File Sharing.

Console Log

The console log shows a simplified output of sent and received requests.


Carrier
11:16 AM
Console
[1] StartSession (request)
11:16:26.11
[2] StartSession (response)
11:16:26.13
[1] RegisterAppInterface (request)
11:16:26.13
[2] RegisterAppInterface (response)
11:16:26.14 - SUCCESS
[2] Framework Version: 4.1.0
11:16:26.14
[2] OnHMIStatus (notification)
11:16:26.15
[2] OnHMIStatus (notification)
11:16:28.43
[1] Alert (request)
11:16:37.17
[3] OnHMIStatus (notification)
11:16:37.17
[2] Alert (response)
11:16:42.18 - SUCCESS
RPCs
Modules
Console

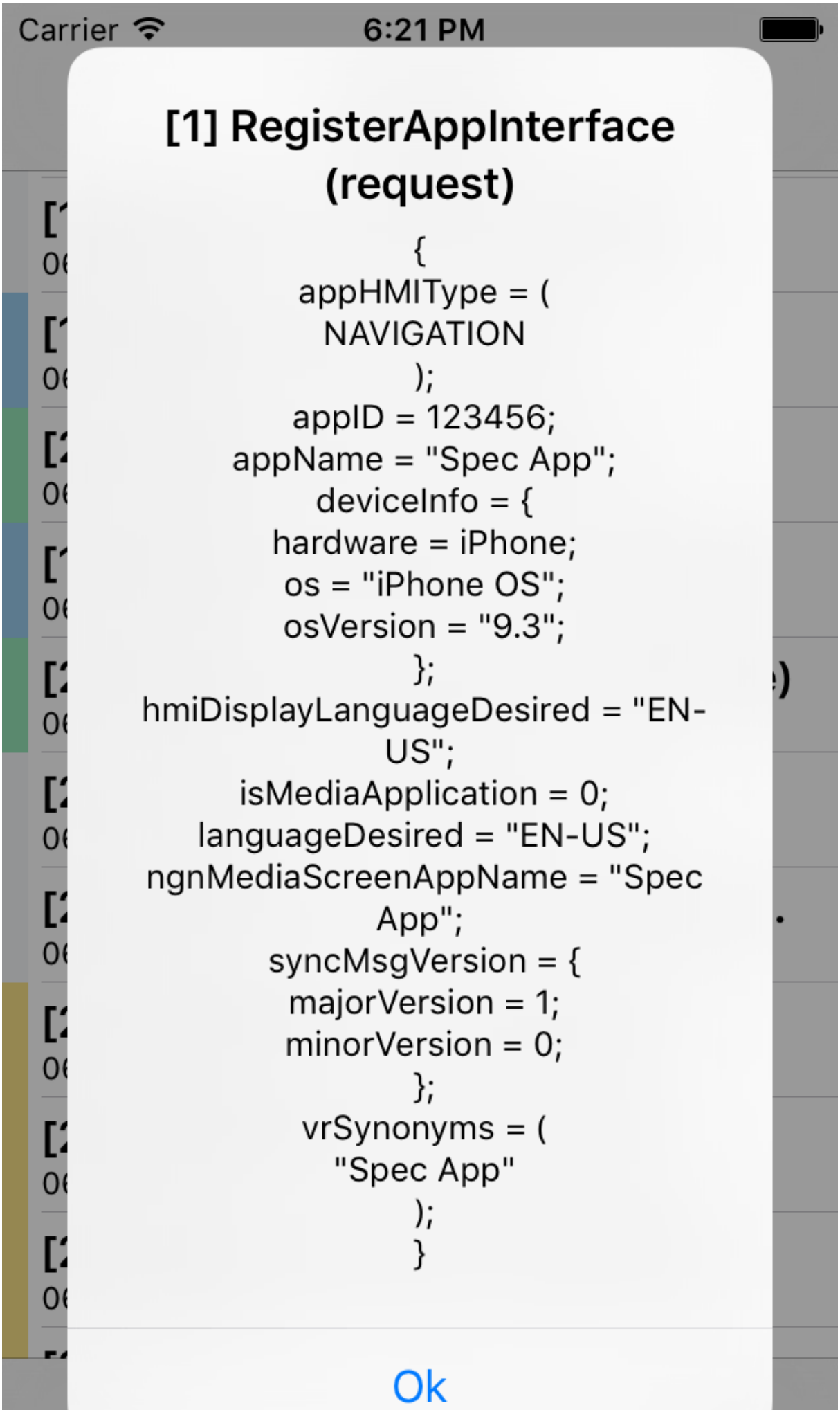
CONSOLE COLOR CODES

The console logs are color coded for quick identification.

1. **White** - Used for logs with no additional data.
2. **Blue** - Used for requests sent to the SDL Core.
3. **Green** - Used for responses from the SDL Core. There are three possible response types:
    - *Successful*: these response types are colored green.
    - *Aborted, Timed-Out, or Warnings*: these response types are colored yellow.
    - *Miscellaneous*: these response types are colored red.
4. **Yellow** - Used for notifications sent from the SDL Core.

RPC JSON

Tapping once on a RPC call in the console will reveal the JSON associated with that RPC call, if applicable.


Carrier
6:21 PM
[1] RegisterAppInterface (request)
{
appHMIType = (
NAVIGATION
);
appID = 123456;
appName = "Spec App";
deviceInfo = {
hardware = iPhone;
os = "iPhone OS";
osVersion = "9.3";
};
hmiDisplayLanguageDesired = "EN-US";
isMediaApplication = 0;
languageDesired = "EN-US";
ngnMediaScreenAppName = "Spec App";
syncMsgVersion = {
majorVersion = 1;
minorVersion = 0;
};
vrSynonyms = (
"Spec App"
);
}
Ok

A Special Note About Putfile

A `putFile` is the RPC responsible for sending binary data from our mobile libraries to the SDL Core. The RPC Builder app provides support for adding any type of file: either from the camera roll (for images) or iTunes shared storage for any other kind of files. Similar to adding custom RPC spec files, any file located within the `BulkData` directory will be present in local storage and be usable for upload.

Need Help?

If you need general assistance, or have other questions, you can sign up for the SDL Slack and chat with other developers and the maintainers of the project.

Found a Bug?

If you see a bug, feel free to post an issue.

Want to Help?

If you want to help add more features, please file a pull request.

Updating from 4.2 and below to 4.3+

This guide is used to show the update process for a developer using a version before 4.3, using the `SDLProxy` to using the new `SDLManager` class available in 4.3 and newer. Although this is not a breaking change, v4.3+ makes significant deprecations and additions that will simplify your code. For our examples through this guide, we are going to be using the version 1.0.0 of the Hello SDL project.

You can download this version here.

Updating the Podfile

For this guide, we will be using the most recent version of SDL at this time: 4.5.5. To change the currently used version, you can open up the `Podfile` located in the root of the `hello_sdl_ios-1.0.0` directory.

Change the following line

```
pod 'SmartDeviceLink-iOS', '~> 4.2.3'
```

to

```
pod 'SmartDeviceLink-iOS', '~> 4.5.5'
```

You may then be able to run `pod install` to install this version of SDL into the app. For more information on how to use Cocoapods, check out the Getting Started > Installation section.

After SDL has been updated, open up the `HelloSDL.xcworkspace`.

You will notice that the project will still compile, but with deprecation warnings.

NOTE

Currently, `SDLProxy` is still supported in versions but is deprecated, however in the future this accessibility will be removed.

Response and Event Handlers

A big change with migration to versions of SDL 4.3 and later is the change from a delegate-based to a notification-based and/or completion-handler based infrastructure. All delegate callbacks relating to Responses and Notifications within `SDLProxyListener.h` will now be available as iOS notifications, with their names listed in `SDLNotificationConstants.h`.

We have also added the ability to have completion handlers for when a request's response comes back. This allows you to simply set a response handler when sending a request and be notified in the block when the response returns or fails. Additional handlers are available on certain RPCs that are associated with SDL notifications, such as `SubscribeButton`, when that button is pressed, you will receive a call on the handler.

Because of this, any delegate function will become non-functional when migrating to `SDLManager` from `SDLProxy`, but changing these to use the new

handlers is simple and will be described in the section **SDLProxy to SDLManager**.

Deprecating SDLRPCRequestFactory

If you are using the `SDLRPCRequestFactory` class, you will need to update the initializers of all RPCs to use this. This will be the following migrations:

```
- (void)hsdl_performWelcomeMessage {
    NSLog(@"Send welcome message");
    SDLShow *show = [[SDLShow alloc] init];
    show.mainField1 = WelcomeShow;
    show.alignment = [SDLTextAlignment CENTERED];
    show.correlationID = [self hsdl_getNextCorrelationId];
    [self.proxy sendRPC:show];

    SDLSpeak *speak = [SDLRPCRequestFactory buildSpeakWithTTS:
WelcomeSpeak correlationID:[self hsdl_getNextCorrelationId]];
    [self.proxy sendRPC:speak];
}
```

to

```
- (void)hsdl_performWelcomeMessage {
    NSLog(@"Send welcome message");
    SDLShow *show = [[SDLShow alloc] initWithMainField1:
WelcomeShow mainField2:nil alignment:SDLTextAlignment.CENTERED];
    [self.proxy sendRPC:show];

    SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:WelcomeSpeak];
    [self.proxy sendRPC:speak];
}
```

NOTE

We are not updating all of the functions that utilize `SDLRPCRequestFactory`, because we are going to be deleting those functions later on in the guide.

SDLProxy to SDLManager

In versions following 4.3, the `SDLProxy` is no longer your main point of interaction with SDL. Instead, `SDLManager` was introduced to allow for apps to more easily integrate with SDL, and to not worry about things such as registering their app, uploading images, and showing a lock screen.

Our first step of removing the usage of `SDLProxy` is to add in an `SDLManager` instance variable. `SDLManager` is started with a `SDLConfiguration`, which contains settings relating to the application.

```
@interface HSDLProxyManager () <SDLManagerDelegate> // Replace
SDLProxyListener with SDLManagerDelegate

@property (nonatomic, strong) SDLManager *manager; // New instance
variable
@property (nonatomic, strong) SDLLifecycleConfiguration *
lifecycleConfiguration; // New instance variable
@property (nonatomic, strong) SDLProxy *proxy;
@property (nonatomic, assign) NSUInteger correlationID;

@property (nonatomic, strong) NSNumber *appIconId;

@property (nonatomic, strong) NSMutableSet *remoteImages;

@property (nonatomic, assign, getter=isGraphicsSupported) BOOL
graphicsSupported;
@property (nonatomic, assign, getter=isFirstHmiFull) BOOL firstHmiFull;
@property (nonatomic, assign, getter=isFirstHmiNotNone) BOOL
firstHmiNotNone;
@property (nonatomic, assign, getter=isVehicleDataSubscribed) BOOL
vehicleDataSubscribed;

@end
```

SDLProxyListener to SDLManagerDelegate

`SDLManagerDelegate` is a small protocol that gives back only 2 callbacks, as compared to `SDLProxyListener`'s 67 callbacks. As mentioned before, all of these callbacks from `SDLProxyListener` are now sent out as `NSNotification`s, and the names for these are located in `SDLNotificationConstants.h`. From these delegate changes, we can modify the following functions to use the new `SDLManagerDelegate` callbacks

`onProxyClosed` to `managerDidDisconnect`:

```
- (void)onProxyClosed {
    NSLog(@"SDL Disconnect");

    // Reset state variables
    self.firstHmiFull = YES;
    self.firstHmiNotNone = YES;
    self.graphicsSupported = NO;
    [self.remoteImages removeAllObjects];
    self.vehicleDataSubscribed = NO;
    self.appIconId = nil;

    // Notify the app delegate to clear the lockscreen
    [self hsdl_postNotification:HSDLDisconnectNotification info:nil];

    // Cycle the proxy
    [self disposeProxy];
    [self startProxy];
}
```

to

```
- (void)managerDidDisconnect {
    NSLog(@"SDL Disconnect");

    // Reset state variables
    self.firstHmiFull = YES;
    self.firstHmiNotNone = YES;
    self.graphicsSupported = NO;
    self.vehicleDataSubscribed = NO;

    // Notify the app delegate to clear the lockscreen
    [self hsdl_postNotification:HSDLDisconnectNotification info:nil];
}
```

`onOnHMIStatus:` to `hmiLevel:didChangeToLevel:`

```
- (void)onOnHMIStatus:(SDLOnHMIStatus *)notification {
    NSLog(@"HMIStatus notification from SDL");

    // Send welcome message on first HMI FULL
    if ([[SDLHMILevel FULL] isEqualToEnum:notification.hmiLevel]) {
        if (self.isFirstHmiFull) {
            self.firstHmiFull = NO;
            [self hsdl_performWelcomeMessage];
        }

        // Other HMI (Show, PerformInteraction, etc.) would go here
    }

    // Send AddCommands in first non-HMI NONE state (i.e., FULL,
LIMITED, BACKGROUND)
    if (![[SDLHMILevel NONE] isEqualToEnum:notification.hmiLevel]) {
        if (self.isFirstHmiNotNone) {
            self.firstHmiNotNone = NO;
            [self hsdl_addCommands];

            // Other app setup (SubMenu, CreateChoiceSet, etc.) would go
here
        }
    }
}
```

to

```
- (void)hmiLevel:(SDLHMILevel *)oldLevel didChangeToLevel:(
SDLHMILevel *)newLevel {
    NSLog(@"HMIStatus notification from SDL");

    // Send welcome message on first HMI FULL
    if ([[SDLHMILevel FULL] isEqualToEnum:newLevel]) {
        if (self.isFirstHmiFull) {
            self.firstHmiFull = NO;
            [self hsdl_performWelcomeMessage];
        }

        // Other HMI (Show, PerformInteraction, etc.) would go here
    }

    // Send AddCommands in first non-HMI NONE state (i.e., FULL,
LIMITED, BACKGROUND)
    if (![[SDLHMILevel NONE] isEqualToEnum:newLevel]) {
        if (self.isFirstHmiNotNone) {
            self.firstHmiNotNone = NO;
            [self hsdl_addCommands];

            // Other app setup (SubMenu, CreateChoiceSet, etc.) would go
here
        }
    }
}
```

We can also remove the functions relating to lifecycle management and app icons, as the **Creating the Manager** section of the migration guide will handle this:

- `hsdl_uploadImages`
- `onListFilesResponse:`
- `onPutFileResponse:`
- `hsdl_setAppIcon`

And can also remove the `remoteImages` property from the list of instance variables.

Correlation Ids

We no longer require developers to keep track of correlation ids, as `SDLManager` does this for you. Because of this, you can remove `correlationID` and `applConId` from the list of instance variables.

Because of this, we can remove the `hsdl_getNextCorrelationId` method as well.

> NOTE
>
> If you set the correlation id, it will be overwritten by `SDLManager`.

Creating the Manager

In `HSDLProxyManager`'s `init` function, we will build these components, and begin removing components that are no longer needed, as `SDLManager` handles it.

```
- (instancetype)init {
    if (self = [super init]) {
        _correlationID = 1; // No longer needed, remove instance variable.
        _graphicsSupported = NO;
        _firstHmiFull = YES;
        _firstHmiNotNone = YES;
        _remoteImages = [[NSMutableSet alloc] init]; // No longer needed,
remove instance variable.
        _vehicleDataSubscribed = NO;

        // SDLManager initialization

         // If connecting via USB (to a vehicle).
//        _lifecycleConfiguration = [SDLLifecycleConfiguration
defaultConfigurationWithAppName:AppName appId:AppId];

        // If connecting via TCP/IP (to an emulator).
        _lifecycleConfiguration = [SDLLifecycleConfiguration
debugConfigurationWithAppName:AppName appId:AppId ipAddress:
RemoteIpAddress port:RemotePort];

        _lifecycleConfiguration.appType = AppIsMediaApp ? [
SDLAppHMIType MEDIA] : [SDLAppHMIType DEFAULT];
        _lifecycleConfiguration.shortAppName = ShortAppName;
        _lifecycleConfiguration.voiceRecognitionCommandNames = @[
AppVrSynonym];
        _lifecycleConfiguration.ttsName = [SDLTTSChunk
textChunksFromString:AppName];

        UIImage* appIcon = [UIImage imageNamed:IconFile];
        if (appIcon) {
            _lifecycleConfiguration.appIcon = [SDLArtwork
artworkWithImage:appIcon name:IconFile asImageFormat:
SDLArtworkImageFormatPNG];
        }

        // SDLConfiguration contains the lifecycle and lockscreen
configurations
        SDLConfiguration *configuration = [SDLConfiguration
configurationWithLifecycle:_lifecycleConfiguration lockScreen:[
SDLLockScreenConfiguration enabledConfiguration]];

        _manager = [[SDLManager alloc] initWithConfiguration:
configuration delegate:self];
    }
    return self;
}
```

We must also update the `RemotePort` constant from a type of `NSString *` to `UInt16`.

Because we the way we configure the app's properties via `SDLLifecycleConfiguration` now, we do not need to actually send an `SDLRegisterAppInterface` request. Because of this, we can remove the `onProxyOpened` method and it's contents.

Built-In Lock Screen

Versions of SDL moving forward contain a lock screen manager to allow for easily customizing and using a lock screen. For more information, please check out the Adding the Lock Screen section.

With the lockscreen handles for us now, we can remove the following from `HSDLProxyManager`:

Constants (from .h and .m)
- `HSDLDisconnectNotification`
- `HSDLLockScreenStatusNotification`
- `HSDLNotificationUserInfoObject`

Functions
- `hsdl_postNotification:info:`
- last line of `managerDidDisconnect`
- `onOnLockScreenNotification:`

We also can eliminate the `LockScreenViewController` files, and remove the following line numbers/ranges from `AppDelegate.m`:
- lines 61-121
- lines 25-31
- lines 15-16

We also can open Main.storyboard, and remove the `LockScreenViewController`.

Starting the Manager and Register App Interface

In previous implementations, a developer would need to react to the `onRegisterAppInterfaceResponse:` to get information regarding their application and the currently connected Core. Now, however, we can simply access these properties after the `SDLManager` has been started.

First, we must start the manager. Change the `startProxy` function from:

```
- (void)startProxy {
    NSLog(@"startProxy");

    // If connecting via USB (to a vehicle).
//    self.proxy = [SDLProxyFactory buildSDLProxyWithListener:self];

    // If connecting via TCP/IP (to an emulator).
    self.proxy = [SDLProxyFactory buildSDLProxyWithListener:self tcpIPAddress:RemoteIpAddress tcpPort:RemotePort];
}
```

to:

```
- (void)startProxy {
    NSLog(@"startProxy");

    __weak typeof(self) weakself = self;
    [self.manager startWithReadyHandler:^(BOOL success, NSError *
_Nullable error) {
        if (!success) {
            NSLog(@"Error trying to start SDLManager: %@", error);
            return;
        }

        NSNumber<SDLBool> graphicSupported = weakself.
systemCapabilityManager.displayCapabilities.graphicSupported
        if (graphicSupported != nil) {
            weakself.graphicsSupported = graphicSupported;
        }
    }];
}
```

We can now remove `onRegisterAppInterfaceResponse:`.

Stopping the Manager

Stopping the manager is simply changing from

```
- (void)disposeProxy {
    NSLog(@"disposeProxy");
    [self.proxy dispose];
    self.proxy = nil;
}
```

to

```
- (void)disposeProxy {
    NSLog(@"disposeProxy");
    [self.manager stop];
}
```

Adding Notification Handlers

Registering for a notification is similar to registering for `NSNotification`s. The list of these subscribable notifications is in `SDLNotificationConstants.h`. For this project, we are observing the `onDriverDistraction:` notification and logging a string. We will modify this to instead listen for a notification and the log the same string.

Remove this function

```
- (void)onOnDriverDistraction:(SDLOnDriverDistraction *)notification {
    NSLog(@"OnDriverDistraction notification from SDL");
    // Some RPCs (depending on region) cannot be sent when driver
distraction is active.
}
```

And add in the notification observer

```
- (instancetype)init {
    if (self = [super init]) {
        // Previous code setting up SDLManager

        // Add in the notification observer
        [[NSNotificationCenter defaultCenter] addObserverForName:
SDLDidChangeDriverDistractionStateNotification object:nil queue:nil
usingBlock:^(NSNotification * _Nonnull note) {
            SDLRPCNotificationNotification* notification = (
SDLRPCNotificationNotification*)note;

            if (![notification.notification isKindOfClass:
SDLOnDriverDistraction.class]) {
                return;
            }

            NSLog(@"OnDriverDistraction notification from SDL");
            // Some RPCs (depending on region) cannot be sent when
driver distraction is active.
        }];
    }
    return self;
}
```

We will also remove all of the remaining delegate functions from `SDLProxyListener`, except for `onAddCommandResponse:`.

Handling command notifications

`SDLAddCommand` utilizes the new handler mechanism for responding to when a user interacts with the command you have added. When using the initializer, you can see we set the new `handler` property to use the same code we originally wrote in `onOnCommand:`.

```
- (void)hsdl_addCommands {
    NSLog(@"hsdl_addCommands");
    SDLMenuParams *menuParams = [[SDLMenuParams alloc] init];
    menuParams.menuName = TestCommandName;
    SDLAddCommand *command = [[SDLAddCommand alloc] init];
    command.vrCommands = [NSMutableArray arrayWithObject:
TestCommandName];
    command.menuParams = menuParams;
    command.cmdID = @(TestCommandID);

    [self.proxy sendRPC:command];
}

- (void)onOnCommand:(SDLOnCommand *)notification {
    NSLog(@"OnCommand notification from SDL");

    // Handle sample command when triggered
    if ([notification.cmdID isEqual:@(TestCommandID)]) {
        SDLShow *show = [[SDLShow alloc] init];
        show.mainField1 = @"Test Command";
        show.alignment = [SDLTextAlignment CENTERED];
        show.correlationID = [self hsdl_getNextCorrelationId];
        [self.proxy sendRPC:show];

        SDLSpeak *speak = [SDLRPCRequestFactory buildSpeakWithTTS:
@"Test Command" correlationID:[self hsdl_getNextCorrelationId]];
        [self.proxy sendRPC:speak];
    }
}
```

to

```
- (void)hsdl_addCommands {
    NSLog(@"hsdl_addCommands");
    SDLAddCommand *command = [[SDLAddCommand alloc] initWithId:
TestCommandID vrCommands:@[TestCommandName] menuName:
TestCommandName handler:^(__kindof SDLRPCNotification * _Nonnull
notification) {
        if (![notification isKindOfClass:SDLOnCommand.class]) {
            return;
        }

        NSLog(@"OnCommand notification from SDL");
        SDLOnCommand* onCommand = (SDLOnCommand*)notification;

        // Handle sample command when triggered
        if ([onCommand.cmdID isEqual:@(TestCommandID)]) {
            SDLShow *show = [[SDLShow alloc] initWithMainField1:@"Test
Command" mainField2:nil alignment:SDLTextAlignment.CENTERED];
            [self.proxy sendRPC:show];

            SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:@"Test
Command"];
            [self.proxy sendRPC:speak];
        }
    }];

    [self.proxy sendRPC:command];
}
```

Sending Requests via `SDLManager`

As mentioned in **Response and Event Handlers**, `SDLManager` provides the ability to easily react to responses for RPCs we send out. `SDLManager` has two functions for sending RPCs:
- `sendRequest:withResponseHandler:`
- `sendRequest:`

We will update our `SDLAddCommand` request to both send the request via `SDLManager` instead of `SDLProxy`, as well as react to the response that we get back.

From

```
- (void)hsdl_addCommands {
    NSLog(@"hsdl_addCommands");
    SDLAddCommand *command = [[SDLAddCommand alloc] initWithId:
TestCommandID vrCommands:@[TestCommandName] menuName:
TestCommandName handler:^(__kindof SDLRPCNotification * _Nonnull
notification) {
        if (![notification isKindOfClass:SDLOnCommand.class]) {
            return;
        }

        NSLog(@"OnCommand notification from SDL");
        SDLOnCommand* onCommand = (SDLOnCommand*)notification;

        // Handle sample command when triggered
        if ([onCommand.cmdID isEqual:@(TestCommandID)]) {
            SDLShow *show = [[SDLShow alloc] initWithMainField1:@"Test
Command" mainField2:nil alignment:SDLTextAlignment.CENTERED];
            [self.proxy sendRPC:show];

            SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:@"Test
Command"];
            [self.proxy sendRPC:speak];
        }
    }];

    [self.proxy sendRPC:command];
}
```

to

```
- (void)hsdl_addCommands {
    NSLog(@"hsdl_addCommands");
    SDLAddCommand *command = [[SDLAddCommand alloc] initWithId:
TestCommandID vrCommands:@[TestCommandName] menuName:
TestCommandName handler:^(__kindof SDLRPCNotification * _Nonnull
notification) {
        if (![notification isKindOfClass:SDLOnCommand.class]) {
            return;
        }

        NSLog(@"OnCommand notification from SDL");
        SDLOnCommand* onCommand = (SDLOnCommand*)notification;

        // Handle sample command when triggered
        if ([onCommand.cmdID isEqual:@(TestCommandID)]) {
            SDLShow *show = [[SDLShow alloc] initWithMainField1:@"Test
Command" mainField2:nil alignment:SDLTextAlignment.CENTERED];
            [self.manager sendRequest:show];

            SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:@"Test
Command"];
            [self.manager sendRequest:speak];
        }
    }];

    [self.manager sendRequest:command withResponseHandler:^(
__kindof SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse
 * _Nullable response, NSError * _Nullable error) {
        NSLog(@"AddCommand response from SDL: %@ with info: %@",
response.resultCode, response.info);
    }];
}
```

And now, we can remove `onAddCommandResponse:`

We can also update `hsdl_performWelcomeMessage`

```
- (void)hsdl_performWelcomeMessage {
    NSLog(@"Send welcome message");
    SDLShow *show = [[SDLShow alloc] initWithMainField1:
WelcomeShow mainField2:nil alignment:SDLTextAlignment.CENTERED];
    [self.proxy sendRPC:show];

    SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:WelcomeSpeak];
    [self.proxy sendRPC:speak];
}
```

to

```
- (void)hsdl_performWelcomeMessage {
    NSLog(@"Send welcome message");
    SDLShow *show = [[SDLShow alloc] initWithMainField1:
WelcomeShow mainField2:nil alignment:SDLTextAlignment.CENTERED];
    [self.manager sendRequest:show];

    SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:WelcomeSpeak];
    [self.manager sendRequest:speak];
}
```

Uploading Files via `SDLManager`'s `SDLFileManager`

`SDLPutFile` is the original means of uploading a file. In 4.3+, we have abstracted this out, and instead provide the functionality via two new classes: `SDLFile` and `SDLArtwork`. For more information on these, check out the Uploading Files and Graphics section.

We can change `hsdl_uploadImage:withCorrelationID:` from

```
- (void)hsdl_uploadImage:(NSString *)imageName withCorrelationID:(
NSNumber *)corrId {
    NSLog(@"hsdl_uploadImage: %@", imageName);
    if (imageName) {
        UIImage *pngImage = [UIImage imageNamed:IconFile];
        if (pngImage) {
            NSData *pngData = UIImagePNGRepresentation(pngImage);
            if (pngData) {
                SDLPutFile *putFile = [[SDLPutFile alloc] init];
                putFile.syncFileName = imageName;
                putFile.fileType = [SDLFileType GRAPHIC_PNG];
                putFile.persistentFile = @YES;
                putFile.systemFile = @NO;
                putFile.offset = @0;
                putFile.length = [NSNumber numberWithUnsignedLong:
pngData.length];
                putFile.bulkData = pngData;
                putFile.correlationID = corrId;
                [self.proxy sendRPC:putFile];
            }
        }
    }
}
```

to

```
- (void)hsdl_uploadImage:(NSString *)imageName {
    NSLog(@"hsdl_uploadImage: %@", imageName);
    if (!imageName) {
        return;
    }

    UIImage *pngImage = [UIImage imageNamed:imageName];
    if (!pngImage) {
        return;
    }

    SDLFile *file = [SDLArtwork persistentArtworkWithImage:pngImage
name:imageName asImageFormat:SDLArtworkImageFormatPNG];
    [self.manager.fileManager uploadFile:file completionHandler:^(BOOL
success, NSUInteger bytesAvailable, NSError * _Nullable error) {
        if (!success) {
            NSLog(@"Error uploading file: %@", error);
        }

        NSLog(@"File uploaded");
    }];
}
```

We can now finally remove `SDLProxy` from the project's instance variables.

Updating from v4.3+ to v5.0+

A number of breaking changes have been made to the SDL library in v5.0+. This means that it is unlikely your project will compile without changes.

Changes to SDL Enums

SDLEnums have changed from being objects in SDL v4.X to strings in Obj-C and Enums in Swift. This means that every usage of SDL enums in your app integration will need changes.

OBJ-C

Old:

```
- (void)hmiLevel:(SDLHMILevel *)oldLevel didChangeToLevel:(
SDLHMILevel *)newLevel {
    if (![newLevel isEqualToEnum:[SDLHMILevel NONE]] && (self.
firstTimeState == SDLHMIFirstStateNone)) {
        // This is our first time in a non-NONE state
    }

    if ([newLevel isEqualToEnum:[SDLHMILevel FULL]] && (self.
firstTimeState != SDLHMIFirstStateFull)) {
        // This is our first time in a FULL state
    }

    if ([newLevel isEqualToEnum:[SDLHMILevel FULL]]) {
        // We entered full
    }
}
```

New:

```
- (void)hmiLevel:(SDLHMILevel)oldLevel didChangeToLevel:(
SDLHMILevel)newLevel {
    if (![newLevel isEqualToEnum:SDLHMILevelNone] && (self.
firstTimeState == SDLHMIFirstStateNone)) {
        // This is our first time in a non-NONE state
    }

    if ([newLevel isEqualToEnum:SDLHMILevelFull] && (self.firstTimeState
 != SDLHMIFirstStateFull)) {
        // This is our first time in a FULL state
    }

    if ([newLevel isEqualToEnum:SDLHMILevelFull]) {
        // We entered full
    }
}
```

Note the differences between, e.g. `[SDLHMILevel FULL]` and `SDLHMILevelFull` .

SWIFT

Old: (Swift 3)

```
func hmiLevel(_ oldLevel: SDLHMILevel, didChangeTo newLevel:
SDLHMILevel) {
    // On our first HMI level that isn't none, do some setup
    if newLevel != .none() && firstTimeState == .none {
        // This is our first time in a non-NONE state
    }

    // HMI state is changing from NONE or BACKGROUND to FULL or
LIMITED
    if (newLevel == .full() && firstTimeState != .full) {
        // This is our first time in a FULL state
    }

    if (newLevel == .full()) {
        // We entered full
    }
}
```

New: (Swift 4)

```
func hmiLevel(_ oldLevel: SDLHMILevel, didChangeToLevel:
SDLHMILevel) {
    // On our first HMI level that isn't none, do some setup
    if didChangeToLevel != .none && firstTimeState == .none {
        // This is our first time in a non-NONE state
    }

    // HMI state is changing from NONE or BACKGROUND to FULL or
LIMITED
    if (didChangeToLevel == .full && firstTimeState != .full) {
        // This is our first time in a FULL state
    }

    if (didChangeToLevel == .full {
        // We entered full
    }
}
```

Note the differences between, e.g. `.full()` and `.full`.

Changes to RPC Handlers

Old:

```
SDLSubscribeButton *button = [[SDLSubscribeButton alloc]
initWithHandler:^(__kindof SDLRPCNotification * _Nonnull notification) {
    if (![notification isKindOfClass:[SDLOnButtonPress class]]) {
        return;
    }
    SDLOnButtonPress *buttonPress = (SDLOnButtonPress *)notification;
    if (buttonPress.buttonPressMode != SDLButtonPressMode.SHORT) {
        return;
    }
}];
[manager sendRequest:button];
```

New:

```
SDLSubscribeButton *button = [[SDLSubscribeButton alloc]
initWithHandler:^(SDLOnButtonPress * _Nullable buttonPress,
SDLOnButtonEvent * _Nullable buttonEvent) {
    if (buttonPress != nil && buttonPress.buttonPressMode !=
SDLButtonPressModeShort) {
        return;
    }
}];

[manager sendRequest:button];
```

SWIFT

Old:

```
let button = SDLSubscribeButton { [unowned store] (notification) in
    guard let buttonPress = notification as? SDLOnButtonPress else {
return }
    guard buttonPress.buttonPressMode == .short() else { return }

    // Button was pressed
}!
button.buttonName = .seekleft()
manager.send(request: button)
```

New:

```
let button = SDLSubscribeButton { [unowned store] (press, event) in
    guard press.buttonPressMode == .short() else { return }

    // Button was pressed
}
button.buttonName = .seekLeft
manager.send(request: button)
```

RPC handlers for `SDLAddCommand`, `SDLSoftButton`, and `SDLSubscribeButton` have been altered to provide more accurate notifications within the handler.

SDLConfiguration Changes

`SDLConfiguration`, used to initialize `SDLManager` has changed slightly. When creating a configuration, a logging configuration is now required. Furthermore, if you are creating a video streaming `NAVIGATION` or `PROJECTION` app, you must now create an `SDLStreamingMediaConfiguration` and add it to your `SDLConfiguration` before initializing the `SDLManager`. Additionally, if your app is in Swift, your initialization may have changed.

OBJ-C

```
SDLConfiguration *config = [SDLConfiguration
configurationWithLifecycle:lifecycleConfig lockScreen:[
SDLLockScreenConfiguration enabledConfiguration] logging:[
SDLLogConfiguration debugConfiguration]];
```

SWIFT

```
let configuration: SDLConfiguration = SDLConfiguration(lifecycle:
lifecycleConfiguration, lockScreen: SDLLockScreenConfiguration.
enabledConfiguration(), logging: SDLLogConfiguration())
```

Multiple File Uploads

You can now upload multiple files (such as images) with one method call and be notified when all finish uploading.

OBJ-C

```
// Upload a batch of files with a completion handler when done
[self.sdlManager.fileManager uploadFiles:files completionHandler:^(
NSError * _Nullable error) {
    <#code#>
}];

// Upload a batch of files, being notified in the progress handler when
each completes (returning whether or not to continue uploading), and
a completion handler when done
[self.sdlManager.fileManager uploadFiles:files progressHandler:^BOOL(
SDLFileName * _Nonnull fileName, float uploadPercentage, NSError *
_Nullable error) {
    <#code#>
} completionHandler:^(NSError * _Nullable error) {
    <#code#>
}];
```

SWIFT

```
// Upload a batch of files with a completion handler when done
sdlManager.fileManager.upload(files: softButtonImages) { (error) in
    <#code#>
}

// Upload a batch of files, being notified in the progress handler when each completes (returning whether or not to continue uploading), and a completion handler when done
sdlManager.fileManager.upload(files: softButtonImages, progressHandler: { (fileName, uploadPercentage, error) -> Bool in
    <#code#>
}) { (error) in
    <#code#>
}
```

Logging Changes

For a comprehensive look at logging with SDL iOS 5.0, see the section dedicated to the subject.

Immutable RPC Collections & Generics

In any RPC that has an array, that array will now be immutable. Any array and dictionary will also now expose what it contains via generics.

For example, within `SDLAlert.h`:

```
@property (nullable, strong, nonatomic) NSArray<SDLSoftButton *> * softButtons;
```

Nullability

SDL now exposes nullability tags for all APIs. This primarily means that you no longer need to use the force-unwrap operator `!` in Swift when creating RPCs.

Video Streaming Enhancements

Video streaming has been overhauled in SDL 5.0; SDL now takes care of just about everything for you automatically including HMI changes and app state changes.

When setting your `SDLConfiguration`, you will have to set up an `SDLStreamingMediaConfiguration`.

```
SDLStreamingMediaConfiguration *streamingConfig = [SDLStreamingMediaConfiguration insecureConfiguration];
SDLConfiguration *config = [[SDLConfiguration alloc] initWithLifecycle:lifecycleConfig lockScreen:[SDLLockScreenConfiguration enabledConfiguration] logging:[SDLLogConfiguration debugConfiguration] streamingMedia:streamingConfig];
```

When you have a `NAVIGATION` or `PROJECTION` app and set this streaming configuration, SDL will automatically start the video streaming session on behalf of your app. When you receive the `SDLVideoStreamDidStartNotification`, you're good to go!

For more information about Video Streaming, see the dedicated section.

Touch Manager Delegate Changes

The touch manager delegate calls have all changed and previous delegate methods won't work. If you are streaming video and set the window, the new

callbacks may return the view that was touched, otherwise it will return nil. For example:

```
- (void)touchManager:(SDLTouchManager *)manager
didReceiveSingleTapForView:(UIView *_Nullable)view atPoint:(CGPoint)
point;
```