

# iOS Guides

Document current as of 10/03/2019 11:35 AM.

## 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 four different ways to install the SDL SDK in your project: Accio, 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](https://cocoapods.org) 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 **.xcodproj** 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 **.xcodproj** 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
```

# Accio Installation

You can install this library using [Accio](#), which is based on SwiftPM syntax. Please follow the steps on the Accio README linked above to initialize Accio into your application. Once installed and initialized into your Xcode project, the root directory should contain a Package.swift file.

1. Open the Package.swift file.
2. Add the following line to the dependencies array of your package file. We suggest always using the latest release of the SDL library.

```
.package(url: "https://github.com/smartdevicelink/sdl_ios.git", .upToNextMajor(from: "6.2.0"),
```

## NOTE

Please see [package manifest format](#) to specify dependencies to a specific branch / version of SDL.

3. Add `"SmartDeviceLink"` or `"SmartDeviceLinkSwift"` to the dependencies array in your target. Use `"SmartDeviceLink"` for Objective-C applications and `"SmartDeviceLinkSwift"` for Swift applications.
4. Install the SDK by running `accio install` in the root folder of your project in Terminal.

# 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. Get an App Id

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

To obtain an app id, sign up at [smartdevicelink.com](https://smartdevicelink.com).

## 2. Enable Background Capabilities

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

## 3. Add SDL Protocol Strings

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

### NOTE

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

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

# Integration Basics

## 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

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

NS_ASSUME_NONNULL_BEGIN

@interface ProxyManager ()

@property (nonatomic, strong) SDLManager *sdManager;

@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

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

## SWIFT

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

### 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 = SDLAppHMTypeMedia;
```

### SWIFT

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

---

## ADDITIONAL APP TYPES

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

### OBJECTIVE-C

```
lifecycleConfiguration.additionalAppTypes = @[
    SDLAppHMITypeInfo];
```

### SWIFT

```
lifecycleConfiguration.additionalAppTypes = [.information];
```

## 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];
```

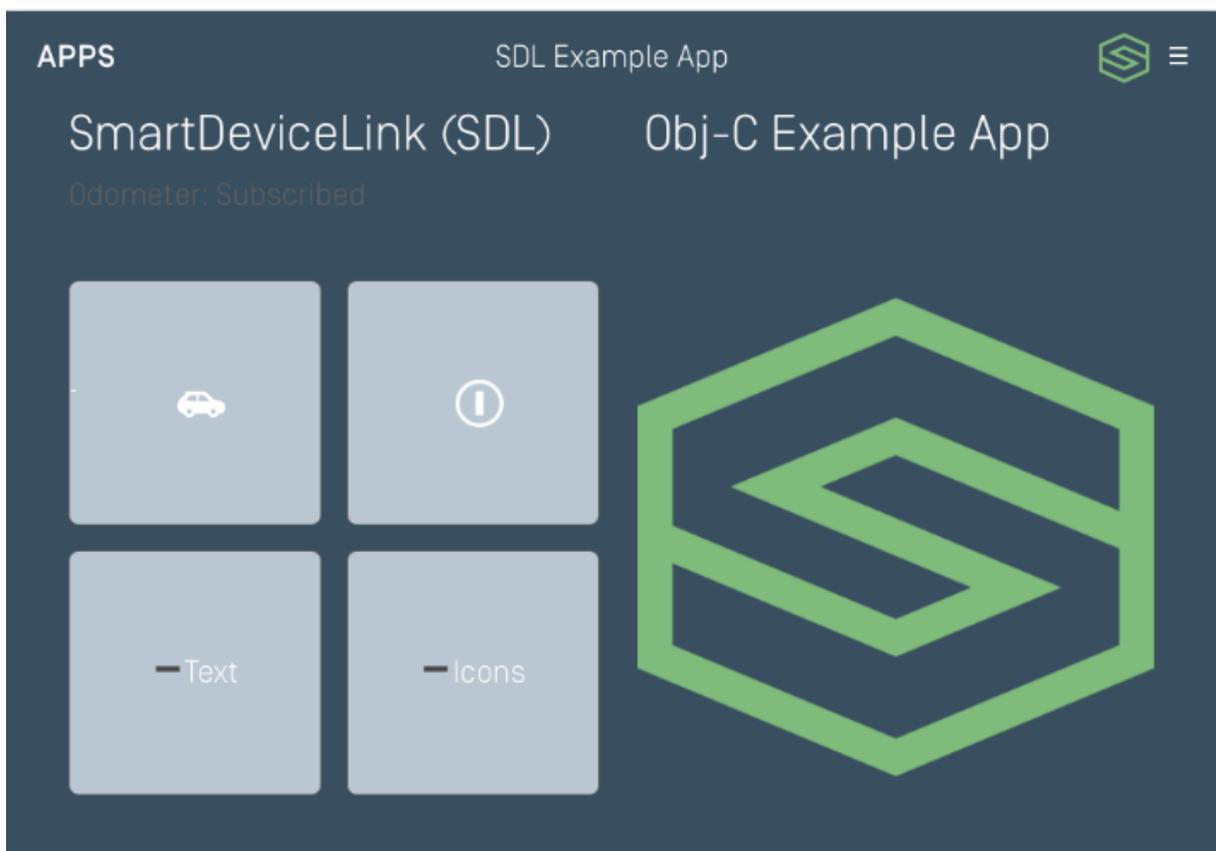
## SWIFT

```
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.

## TEMPLATE COLORING



## 6. Determine SDL Support

You have the ability to determine a minimum SDL protocol and minimum SDL RPC version that your app supports. We recommend not setting these values until your app is ready for production. The OEMs you support will help you configure the correct `minimumProtocolVersion` and `minimumRPCVersion` during the application review process.

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

### OBJECTIVE-C

```
lifecycleConfiguration.minimumProtocolVersion = [SDLVersion  
versionWithString:@"3.0.0"];  
lifecycleConfiguration.minimumRPCVersion = [SDLVersion  
versionWithString:@"4.0.0"];
```

### SWIFT

```
lifecycleConfiguration.minimumProtocolVersion = SDLVersion(string:  
"3.0.0")  
lifecycleConfiguration.minimumRPCVersion = SDLVersion(string: "4.0.0"  
)
```

## 7. Lock Screen

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

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

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

### OBJECTIVE-C

```
[SDLLockScreenConfiguration enabledConfiguration]
```

### SWIFT

```
SDLLockScreenConfiguration.enabled()
```

## 8. Logging

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

## OBJECTIVE-C

```
[SDLLogConfiguration defaultConfiguration]
```

## SWIFT

```
SDLLogConfiguration.default()
```

## 9. File Manager

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

## OBJECTIVE-C

```
[SDLFileManagerConfiguration defaultConfiguration];
```

## SWIFT

```
SDLFileManagerConfiguration.default()
```

## 10. Set the Configuration

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

### 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())
```

## 11. Create a SDLManager

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

### OBJECTIVE-C

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

## SWIFT

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

## 11. 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.

---

## 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](#).

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

```

```

    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

```

SWIFT

```

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)")  
  }  
}
```

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

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

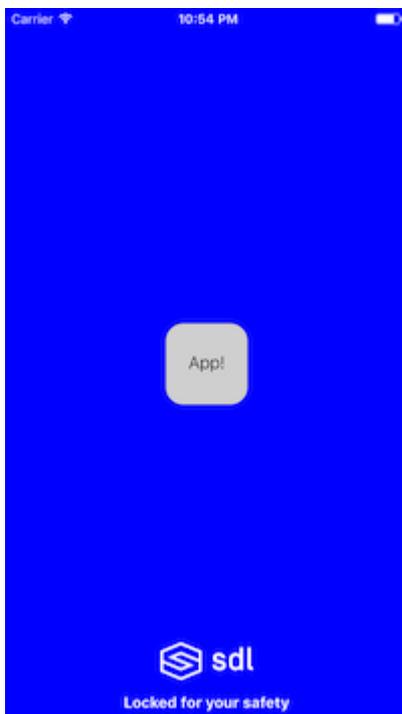
```
SDLLockScreenConfiguration *lockScreenConfiguration = [  
    SDLLockScreenConfiguration enabledConfiguration];
```

## SWIFT

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

# Customizing the Default Lock Screen

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



## Custom Background Color

OBJECTIVE-C

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

## SWIFT

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

## Custom App Icon

### OBJECTIVE-C

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

## SWIFT

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

## Showing the OEM Logo

The default lock screen handles retrieving and setting the OEM logo from head units that support this feature. Currently this feature can not be disabled on the default lock screen.

# Creating a Custom Lock Screen

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

## 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)
```

# Customizing the Lock Screen State

## Disabling the Lock Screen

Please note that a lock screen will be required by most OEMs. You can disable the lock screen manager, but you will then be required to implement your own

logic for showing and hiding the lock screen. This is not recommended as the `SDLLockScreenConfiguration` adheres to most OEM lock screen requirements. However, if you must create a lock screen manager from scratch, the library's lock screen manager can be disabled via the `SDLLockScreenConfiguration` as follows:

## OBJECTIVE-C

```
SDLLockScreenConfiguration *lockScreenConfiguration = [  
    SDLLockScreenConfiguration enabledConfiguration];  
lockScreenConfiguration.enableAutomaticLockScreen = NO;
```

## SWIFT

```
let lockScreenConfiguration = SDLLockScreenConfiguration.  
    enabledConfiguration()  
lockScreenConfiguration.enableAutomaticLockScreen = false
```

## Making the Lock Screen Always On

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

## OBJECTIVE-C

```
SDLLockScreenConfiguration *lockScreenConfiguration = [  
    SDLLockScreenConfiguration enabledConfiguration];  
lockScreenConfiguration.showInOptionalState = YES;
```

## SWIFT

```
let lockScreenConfiguration = SDLLockScreenConfiguration.  
enabledConfiguration()  
lockScreenConfiguration.showInOptionalState = true
```

# Adapting to the Head Unit Language

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

## Setting the Default Language

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

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

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

## Checking the Current Head Unit Language

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

### OBJECTIVE-C

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

### SWIFT

```
let headUnitLanguage = sdIManager.registerResponse?.language  
let headUnitHMIDisplayLanguage = sdIManager.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

```
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 the RPCs you want. If your app does not have the required permissions, requests will be rejected. There are three important things to remember in regards to permissions:

1. You may not be able to send a RPC when the SDL app is closed, in the background, or obscured by an alert. Each RPC has a set of `hmiLevels` 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 to 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 <code>appType</code> of <code>MEDIA</code> or <code>NAVIGATION</code> ). 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 recommend 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

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

## Checking Current Permissions of a Single RPC

### OBJECTIVE-C

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

## SWIFT

```
let observerId = sdIManager.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.sdIManager.permissionManager removeObserverForIdentifier:  
observerId];
```

```
sdIManager.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 occurring between what you are playing, if anything, and some system level audio or navigation application audio.
NOT_AUDIBLE	Your streaming audio is not audible. This could occur during a <code>VRSESSION</code> 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.

# Troubleshooting

## TCP Debug Transport

1. Make sure the correct IP address and port number is set in the `SDLLifecycleConfiguration`.
2. Make sure the device and the SDL emulator are on the same network.
3. If you are running an SDL Core emulator on a virtual machine, and you are using port forwarding to connect your device to the virtual machine, the IP address should be the IP address of your machine hosting the VM, not the IP address of the VM. The port number will be `12345`.
4. Make sure there is no firewall blocking the incoming port `12345` on the machine or VM running the SDL Core emulator. Also make sure your firewall allows that outgoing port.
5. Due to iOS background restrictions, when the app on the device is backgrounded, the app will be unable to communicate with the SDL emulator. This will work on IAP connections.
6. Audio will not play when using a TCP connection. Only IAP connections are currently able to play audio because this happens over the standard Bluetooth / USB system audio channel.

## iAP Production Transport

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

---

## IAP BLUETOOTH PRODUCTION TRANSPORT

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

# 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` response RPC will be sent by Core with the `displayCapabilities`, `buttonCapabilities`, `speechCapabilities` and other properties. You should use this information to create the user interface of your SDL app.

You may access these properties on the `SDLManager.systemCapabilityManager` instance as of library v.6.0.

# System Capability Manager Properties

PARAMETERS	DESCRIPTION	NOTES
displayCapabilities	Information about the HMI display. This includes information about available templates, whether or not graphics are supported, and a list of all text fields and the max number of characters allowed in each text field.	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
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 <a href="#">SDLPrerecordedSpeech.h</a> 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 <a href="#">SDLVRCapabilities.h</a> for more information
audioPassThruCapabilities	Describes the sampling rate, bits per sample, and audio types available.	Check <a href="#">SDLAudioPassThruCapabilities.h</a> for more information
pcmStreamCapabilities	Describes different audio type configurations for the audio PCM stream service, e.g. {8kHz,8-bit,PCM}.	Check <a href="#">SDLAudioPassThruCapabilities.h</a> for more information
hmiCapabilities	Returns whether or not the app can support built-in navigation and phone calls.	Check <a href="#">SDLHMICapabilities.h</a> for more information
appServicesCapabilities	Describes the capabilities of app services including what service types are supported and the current state of services.	Check <a href="#">SDLAppServicesCapabilities.h</a> for more information
navigationCapability	Describes the built-in vehicle navigation system's APIs.	Check <a href="#">SDLNavigationCapability.h</a> for more information
phoneCapability	Describes the built-in phone calling capabilities of the IVI system.	Check <a href="#">SDLPhoneCapability.h</a> for more information
videoStreamingCapability	Describes the abilities of the head unit to video stream projection applications.	Check <a href="#">SDLVideoStreamingCapability.h</a> for more information

PARAMETERS	DESCRIPTION	NOTES
remoteControlCapability	Describes the abilities of an app to control built-in aspects of the IVI system.	Check SDLRemoteControlCapabilities.h for more information

## The Register App Interface Response (RAIR) 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 system capability manager returned by the `RegisterAppInterface` response. Each property is optional, so you may not get data for all the parameters in the following table.

PARAMETERS	DESCRIPTION	NOTES
syncMsgVersion	Specifies the version number of the SmartDeviceLink protocol that is supported by the mobile application.	Check <code>SDLSyncMsgVersion.h</code> for more information
language	The currently active VR +TTS language on the module.	Check <code>SDLLanguage.h</code> for more information
vehicleType	The make, model, year, and the trim of the vehicle.	Check <code>SDLVehicleType.h</code> 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 <code>SDLDiagnosticMessage.h</code> 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 TYPES

Images may be formatted as PNG, JPEG, or BMP. Each `SDLManager.registerResponse.displayCapabilities.imageFields` property will have a list of `imageTypeSupported`.

---

## IMAGE SIZES

If an image is uploaded that is larger than the supported size, that image will be scaled down by Core. All image sizes are available from the `SDLManager.registerResponse.displayCapabilities.imageFields` property once the manager has started successfully.

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

IMAGE NAME	USED IN RPC	DETAILS	HEIGHT	WIDTH	TYPE
cmdIcon	AddCommand	Image shown for commands in the "More..." menu	35px	35px	png, jpg, bmp
applIcon	SetApplication	Image shown as icon in the "Mobile Apps" menu	70px	70px	png, jpg, bmp
graphic	Show	Image shown on the base screen as cover art	185px	185px	png, jpg, bmp

## Querying the RAIR Capabilities

### OBJECTIVE-C

```
BOOL vrCapability = self.sdIManager.systemCapabilityManager.vrCapability;
```

### SWIFT

```
let vrCapability = sdIManager.systemCapabilityManager.vrCapability;
```

# System Capabilities

Most head units provide features that your app can use: making and receiving phone calls, an embedded navigation system, video and audio streaming, as well as supporting app services. To find out if the head unit supports a feature as well as more information about the feature, use the `SDLSystemCapabilityManager` to query the head unit for the desired capability. Querying for capabilities is only available on head units supporting SDL Core v4.5 or greater; if connecting to older head units, the query will return `nil` even if the head unit may support the capability.

## Querying for System Capabilities

OBJECTIVE-C

```
[self.sdlManager.systemCapabilityManager updateCapabilityType:
SDLSystemCapabilityTypeVideoStreaming completionHandler:^(NSError
*_Nullable error, SDLSystemCapabilityManager *_Nonnull
systemCapabilityManager) {
    if (error != nil || systemCapabilityManager.videoStreamingCapability
== nil) {
        return;
    }
    <#Use the video streaming capability#>
}];
```

## SWIFT

```
sdIManager.systemCapabilityManager.updateCapabilityType(.  
videoStreaming) { (error, manager) in  
    guard error == nil, let videoStreamingCapability = manager.  
videoStreamingCapability else {  
        return  
    }  
    <#Use the video streaming capability#>  
}
```

## Subscribing to System Capability Updates

In addition getting the current system capabilities, it is also possible to register to get updates when the head unit capabilities change. To get these notifications you must register using a `subscribeToCapabilityType:` method. This feature is only available on head units supporting v.5.1 or greater.

---

### CHECKING IF THE HEAD UNIT SUPPORTS SUBSCRIPTIONS

#### OBJECTIVE-C

```
BOOL supportsSubscriptions = self.sdIManager.  
systemCapabilityManager.supportsSubscriptions;
```

#### SWIFT

```
let supportsSubscriptions = sdIManager.systemCapabilityManager.  
supportsSubscriptions;
```

---

## SUBSCRIBE TO A CAPABILITY

### OBJECTIVE-C

```
// Subscribing to a capability via a selector callback
[self.sdlManager.systemCapabilityManager subscribeToCapabilityType:
SDLSystemCapabilityTypeNavigation withObserver:self selector:
@selector(navigationCapabilitySelectorCallback:)];

// This can either have one or zero parameters. If one parameter it
must be of type `SDLSystemCapability`. See the [reference
documentation](https://smartdevicelink.com/en/docs/iOS/master/
Classes/SDLSystemCapabilityManager/) for more details (// TODO:
Update with the link to the actual method)
- (void)navigationCapabilitySelectorCallback:(SDLSystemCapability *)
capability {
    SDLNavigationCapability *navCapability = capability.
navigationCapability;

    <#Use the capability#>
}

// Subscribing to a capability via a block callback
[self subscribeToCapabilityType:SDLSystemCapabilityTypeNavigation
usingBlock:^(SDLSystemCapability * _Nonnull capability) {
    SDLNavigationCapability *navCapability = capability.
navigationCapability;
    <#Use the capability#>
}];
```

## SWIFT

```
// Subscribing to a capability via a selector callback
sdIManager.systemCapabilityManager.subscribe(toCapabilityType: .
navigation, withObserver: self, selector: #selector(
navigationCapabilitySelectorCallback(_:)))

@objc private func navigationCapabilitySelectorCallback(_ capability:
SDLSystemCapability) {
    let navCapability = capability.navigationCapability;

    <#Use the capability#>
}

// Subscribing to a capability via a block callback
sdIManager.systemCapabilityManager.subscribe(toCapabilityType: .
navigation) { (capability) in
    let navCapability = capability.navigationCapability;
    <#Use the capability#>
}
```

# Main Screen Templates

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

## Change the Template

To change a template at any time, send a `SetDisplayLayout` RPC to Core.

## 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

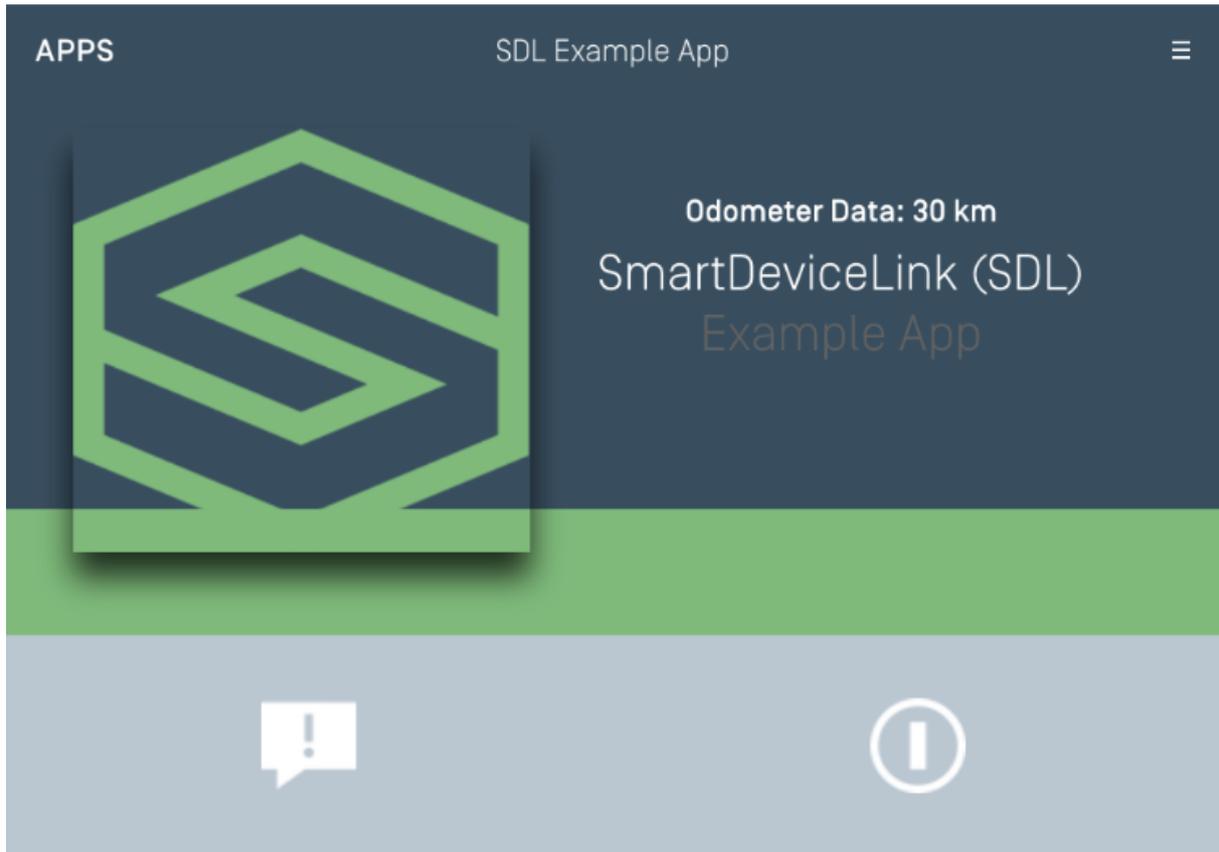
```
let display = SDLSetDisplayLayout(predefinedLayout: .graphicWithText)
sdlManager.send(request: display) { (request, response, error) in
    if response?.resultCode == .success {
        // The template has been set successfully
    }
}
```

## Available Templates

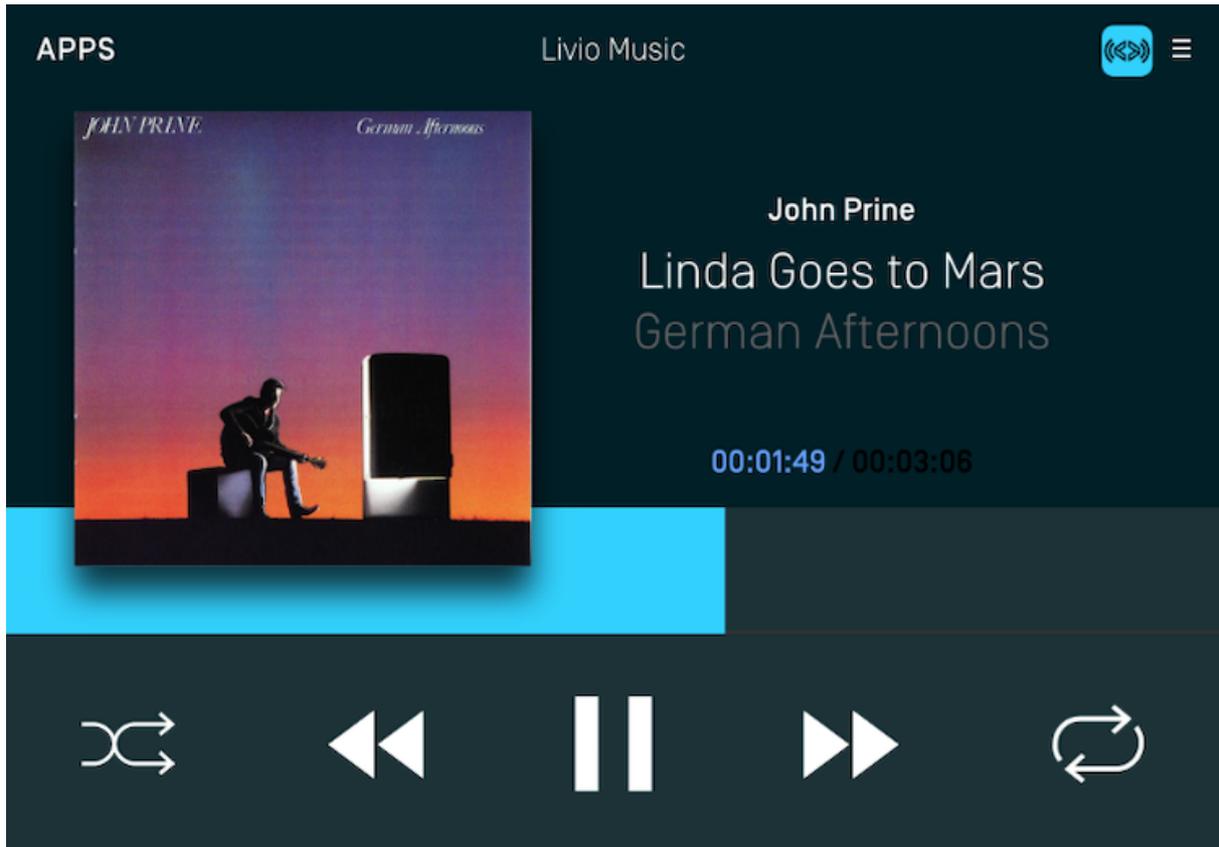
There are fifteen standard templates to choose from, however some head units may only support a subset of these templates. The following examples show how templates will appear on the [Generic HMI](#) and [Ford's SYNC 3 HMI](#).

---

MEDIA



## MEDIA (WITH A PROGRESS BAR)



---

## NON-MEDIA



---

## GRAPHIC WITH TEXT



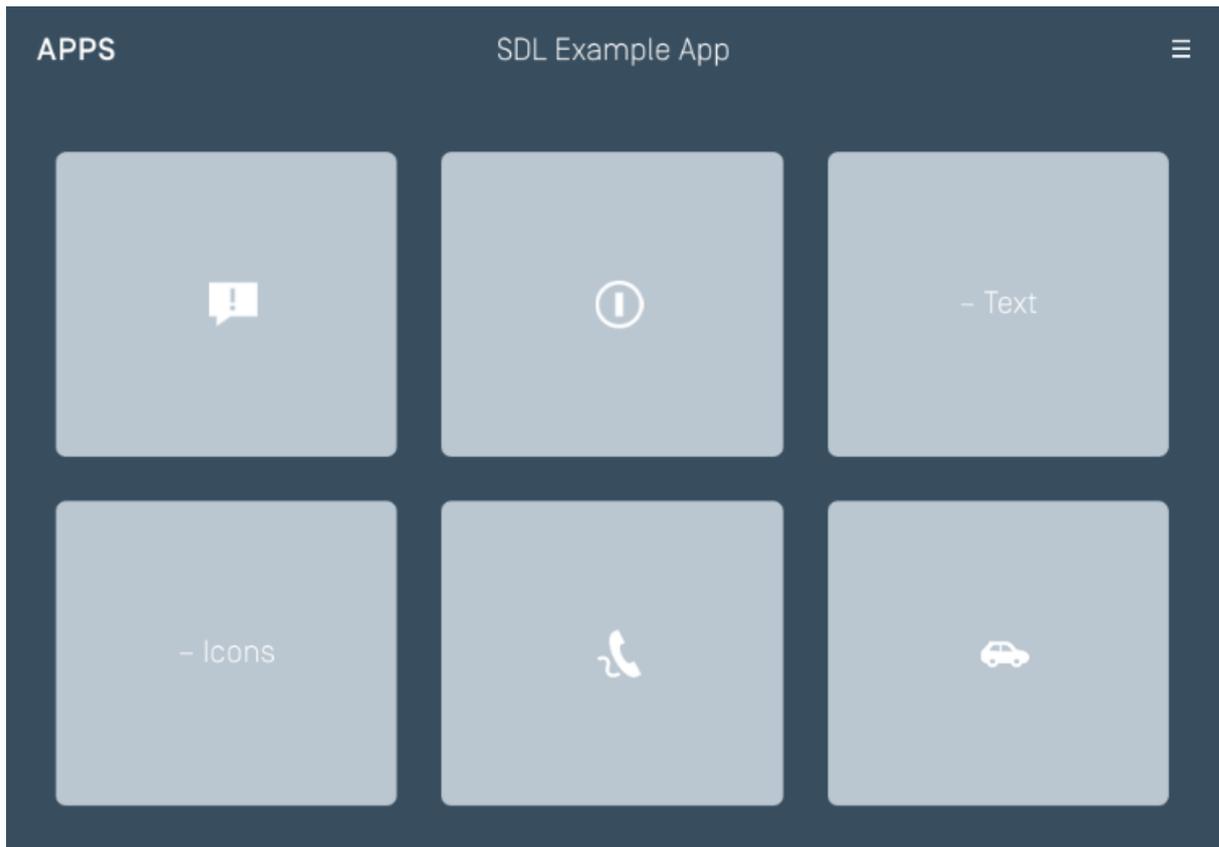
---

## TEXT WITH GRAPHIC



---

## TILES ONLY



---

## GRAPHIC WITH TILES

SYNC 3 - Graphic with Tiles

---

## TILES WITH GRAPHIC

SYNC 3 - Tiles with Graphic

---

## GRAPHIC WITH TEXT AND SOFT BUTTONS

SYNC 3 - Graphic with Text and Soft Buttons

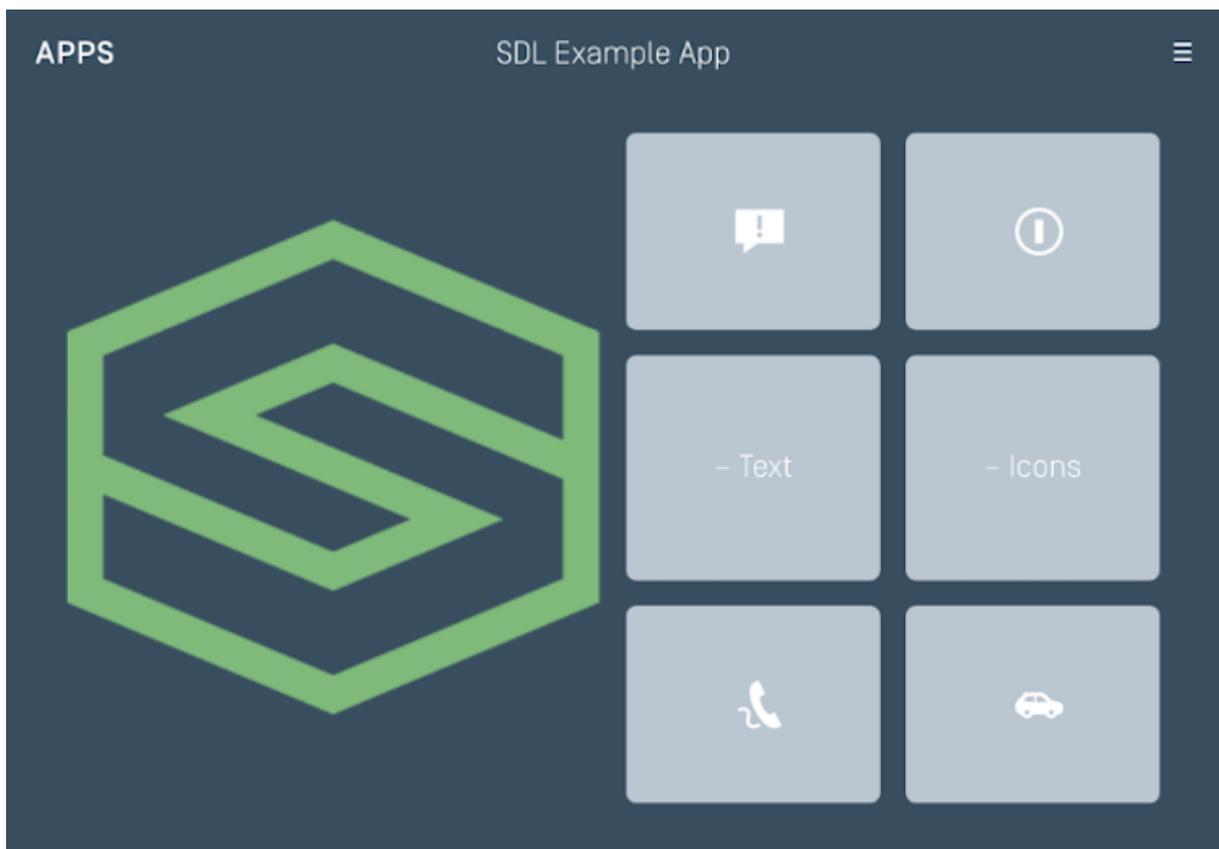
---

## TEXT AND SOFT BUTTONS WITH GRAPHIC

SYNC 3 Text and Softbuttons with Graphic

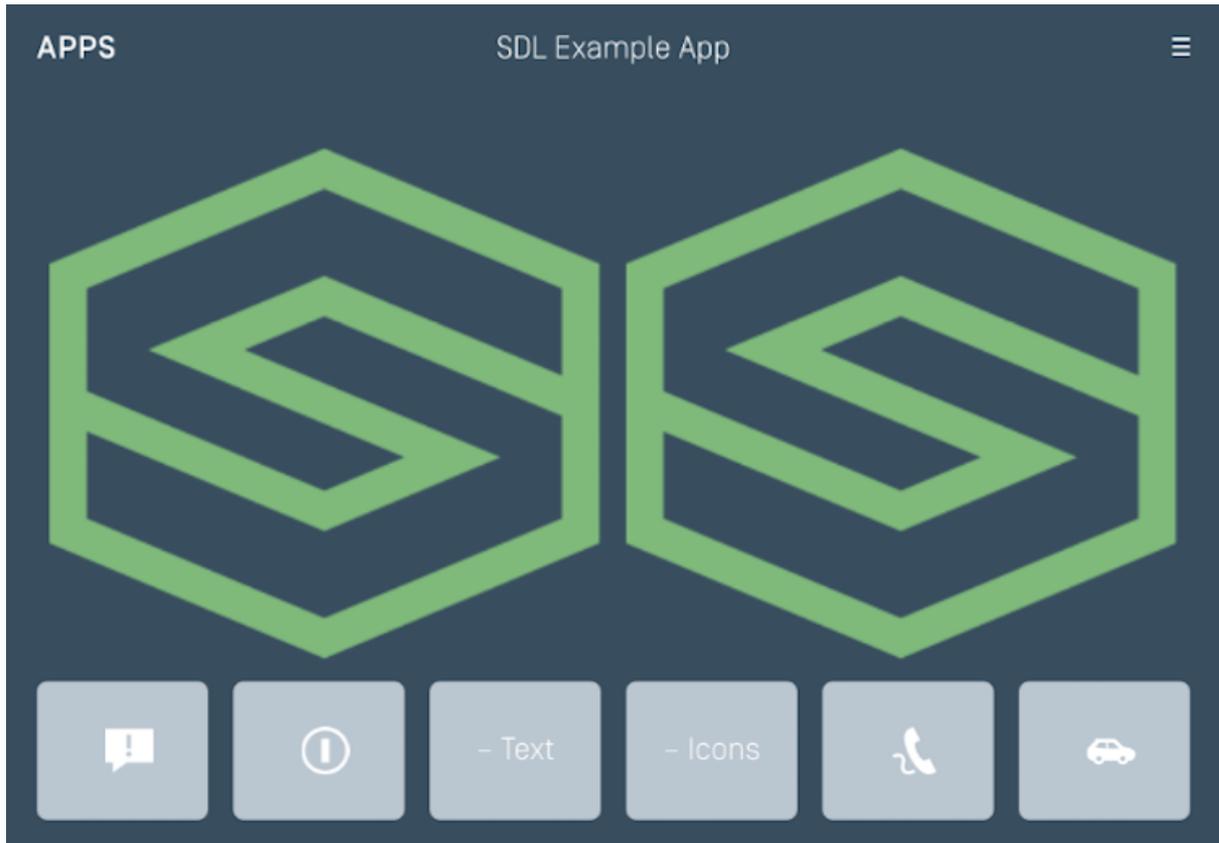
---

## GRAPHIC WITH TEXT BUTTONS



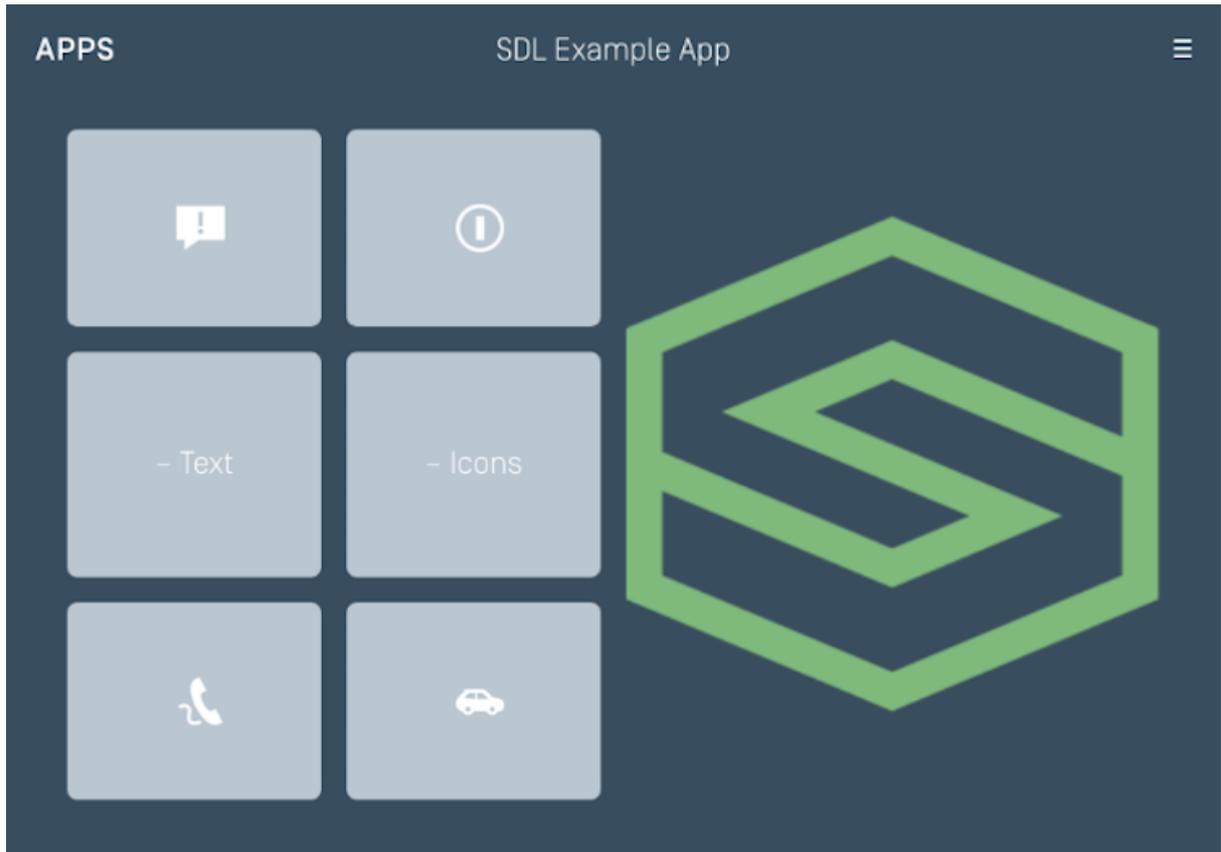
---

## DOUBLE GRAPHIC WITH SOFT BUTTONS



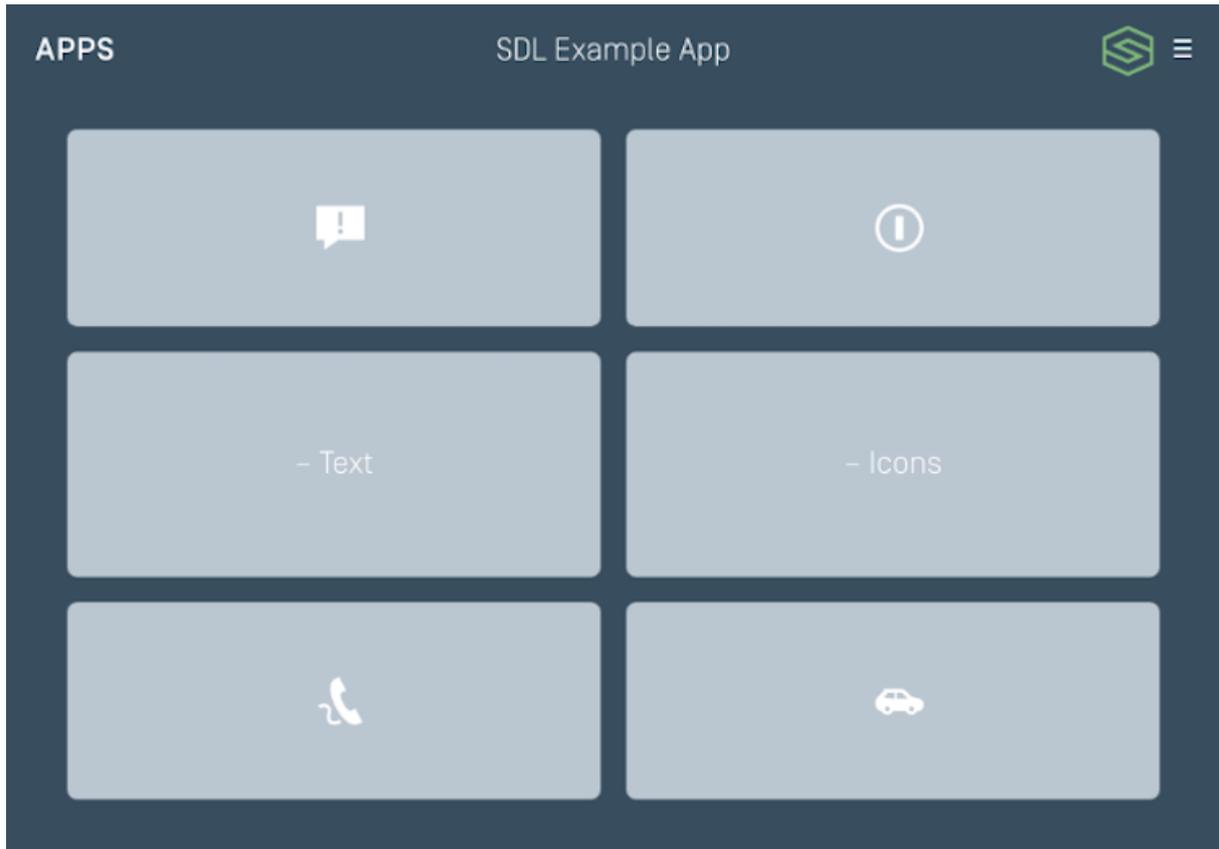
---

## TEXT BUTTONS WITH GRAPHIC



---

## TEXT BUTTONS ONLY



---

## LARGE GRAPHIC WITH SOFT BUTTONS



---

LARGE GRAPHIC ONLY



## Text, Images, and Buttons

This guide covers presenting text and images on the screen as well as creating, showing, and responding to custom buttons you create.

# Template Fields

The `SDLScreenManager` is a manager for easily creating 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 <code>textField1</code>
textField2Type	The type of data provided in <code>textField2</code>
textField3Type	The type of data provided in <code>textField3</code>
textField4Type	The type of data provided in <code>textField4</code>

## 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

```
sdIManager.screenManager.beginUpdates()

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

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

## Removing Text and Images

After you have displayed text and graphics onto the screen, you may want to remove those from being displayed. In order to do so, you only need to set the screen manager property to `nil`.

### OBJECTIVE-C

```
self.sdIManager.screenManager.textField1 = nil;
self.sdIManager.screenManager.textField2 = nil;
self.sdIManager.screenManager.primaryGraphic = nil;
```

## SWIFT

```
sdIManager.screenManager.textField1 = nil  
sdIManager.screenManager.textField2 = nil  
sdIManager.screenManager.primaryGraphic = nil
```

# Soft Button Objects

To create a soft button using the `ScreenManager`, 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. Soft Buttons can contain images, text or both.



## Soft Button Layouts

---

SOFT BUTTON (TEXT ONLY)

## OBJECTIVE-C

```
SDLSoftButtonState *textState = [[SDLSoftButtonState alloc]
initWithStateName:@"<#State Name#>" text:@"<#Button Label
Text#>" image:nil];

SDLSoftButtonObject *softButton = [[SDLSoftButtonObject alloc]
initWithName:@"<#Button Name#>" state:textState handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_NonNullable buttonEvent) {
    if (buttonPress == nil) { return; }
    <#Button selected#>
}];

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

## SWIFT

```
let textState = SDLSoftButtonState(stateName: "<#State Name#>",
text: "<#Button Label Text#>", image: nil)

let softButton = SDLSoftButtonObject(name: "<#Button Name#>",
state: textState) { (buttonPress, buttonEvent) in
    guard buttonPress != nil else { return }
    <#Button selected#>
}

sdlManager.screenManager.softButtonObjects = [softButton]
```

---

## SOFT BUTTON (IMAGE ONLY)

To see if soft buttons support images you should check the `softButtonCapabilities` property on `SDLManager`'s `systemCapabilityManager`.

## OBJECTIVE-C

```
// Check to see if soft buttons support images
BOOL softButtonsSupportImages = self.sdlManager.
systemCapabilityManager.softButtonCapabilities.firstObject.
imageSupported.boolValue;

// If HMI supports images create a soft button with an image
SDLSoftButtonState *imageState = [[SDLSoftButtonState alloc]
initWithStateName:@"<#State Name#>" text:nil image:[[UIImage
imageName:@"<#Image Name#>"] imageWithRenderingMode:<#
UIImageRenderingMode#>]];

SDLSoftButtonObject *softButton = [[SDLSoftButtonObject alloc]
initWithName:@"<#Button Name#>" state:imageState handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_Nonnull buttonEvent) {
    if (buttonPress == nil) { return; }
    <#Button selected#>
}];

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

## SWIFT

```
// Check to see if soft buttons support images
let supportsImages = sdlManager.systemCapabilityManager.
softButtonCapabilities?.first?.imageSupported.boolValue ?? false

// If HMI supports images create a soft button with an image
let imageState = SDLSoftButtonState(stateName: "State Name", text:
"<#State Name#>", image: UIImage(named:"<#Image Name#>")?.
withRenderingMode(<#RenderingMode#>))

let softButton = SDLSoftButtonObject(name: "<#Button Name#>",
state: imageState) { (buttonPress, buttonEvent) in
    guard buttonPress != nil else { return }
    <#Button selected#>
}

sdlManager.screenManager.softButtonObjects = [softButton]
```

---

## SOFT BUTTON (IMAGE AND TEXT)

### OBJECTIVE-C

```
SDLSoftButtonState *state = [[SDLSoftButtonState alloc]
initWithStateName:@"<#State Name#>" text:@"<#Button Label
Text#>" image:[[UIImage imageNamed:@"<#Image Name#>"]
imageWithRenderingMode:<#UIImageRenderingMode#>]];

SDLSoftButtonObject *softButton = [[SDLSoftButtonObject alloc]
initWithName:@"<#Button Name#>" state:state handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_Nullable buttonEvent) {
    if (buttonPress == nil) { return; }
    <#Button selected#>
}];

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

### SWIFT

```
let state = SDLSoftButtonState(stateName: "<#State Name#>", text:
"<#Button Label Text#>", image: UIImage(named:"<#Image
Name#>")?.withRenderingMode(<#RenderingMode#>))

let softButton = SDLSoftButtonObject(name: "<#Button Name#>",
state: state) { (buttonPress, buttonEvent) in
    guard buttonPress != nil else { return }
    <#Button selected#>
}

sdlManager.screenManager.softButtonObjects = [softButton]
```

## Updating the Soft Button State

When the soft button state needs to be updated, simply tell the `SoftButtonObject` to transition to the next state. If your button states do not cycle in a predictable order, you can also tell the soft button 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 screen manager a new array of soft buttons. To delete all soft buttons, simply pass the screen manager an empty array.

## OBJECTIVE-C

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

## SWIFT

```
sdlManager.screenManager.softButtonObjects = []
```

# Templating Images

When connected to a remote system running SDL Core 5.0+, you may be able to use template images. Templated images are tinted by Core so the image is visible regardless of whether your user has set the head unit to day or night mode. For example, if a head unit is in night mode with a dark theme (see [Template Coloring in the Integration Basics section](#) for more details on how to customize theme colors), then your templated images will be displayed as white. In the day theme, the image will automatically change to black.

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

---

## TEMPLATED IMAGES EXAMPLE

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

NIGHT MODE

APPS Livio Music



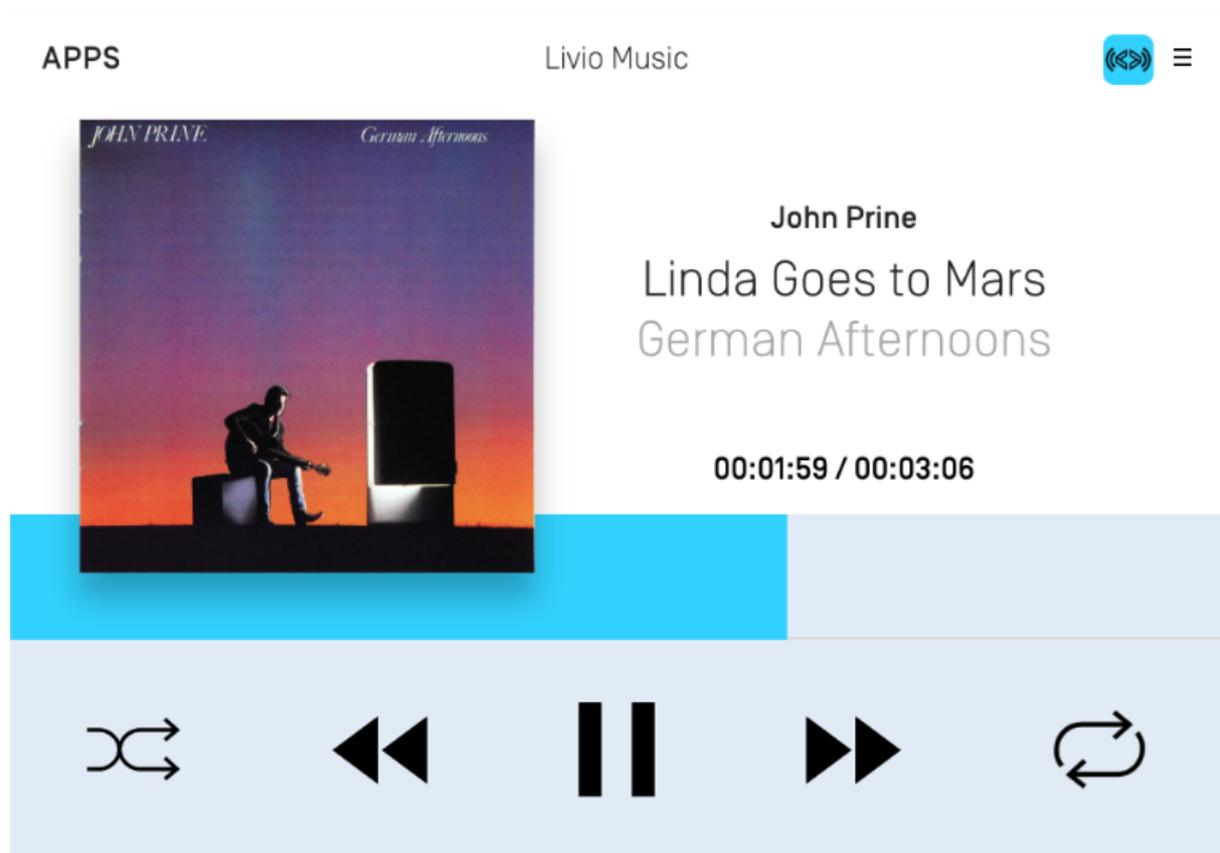
JOHN PRINE German Afternoons

John Prine  
Linda Goes to Mars  
German Afternoons

00:01:49 / 00:03:06



## DAY MODE



## OBJECTIVE-C

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

## SWIFT

```
let image = UIImage(named: "<#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 are fully supported by the screen manager via an `SDLArtwork` initializer.

Static icons can be used in primary and secondary graphic fields, soft button image fields, and menu icon fields.

## OBJECTIVE-C

```
SDLArtwork *staticIconArt = [[SDLArtwork alloc] initWithStaticIcon:
SDLStaticIconNameAlbum];
SDLSoftButtonState *softButtonState1 = [[SDLSoftButtonState alloc]
initWithStateName:@"<#Soft Button State Name#>" text:
@"<#Button Label Text#>" artwork:staticIconArt];

<#Set the state into an `SDLSoftButtonObject` and then set the screen
manager array of soft buttons#>
```

## SWIFT

```
let staticIconArt = SDLArtwork(staticIcon: .album)
let softButtonState1 = SDLSoftButtonState(stateName: "<#Soft Button
State Name#>", text: "<#Button Label Text#>", artwork: staticIconArt)

<#Set the state into an `SDLSoftButtonObject` and then set the screen
manager array of soft buttons#>
```

# Using RPCs

If you don't want to use the screen manager, you can just send raw [Show](#) RPC requests to Core.

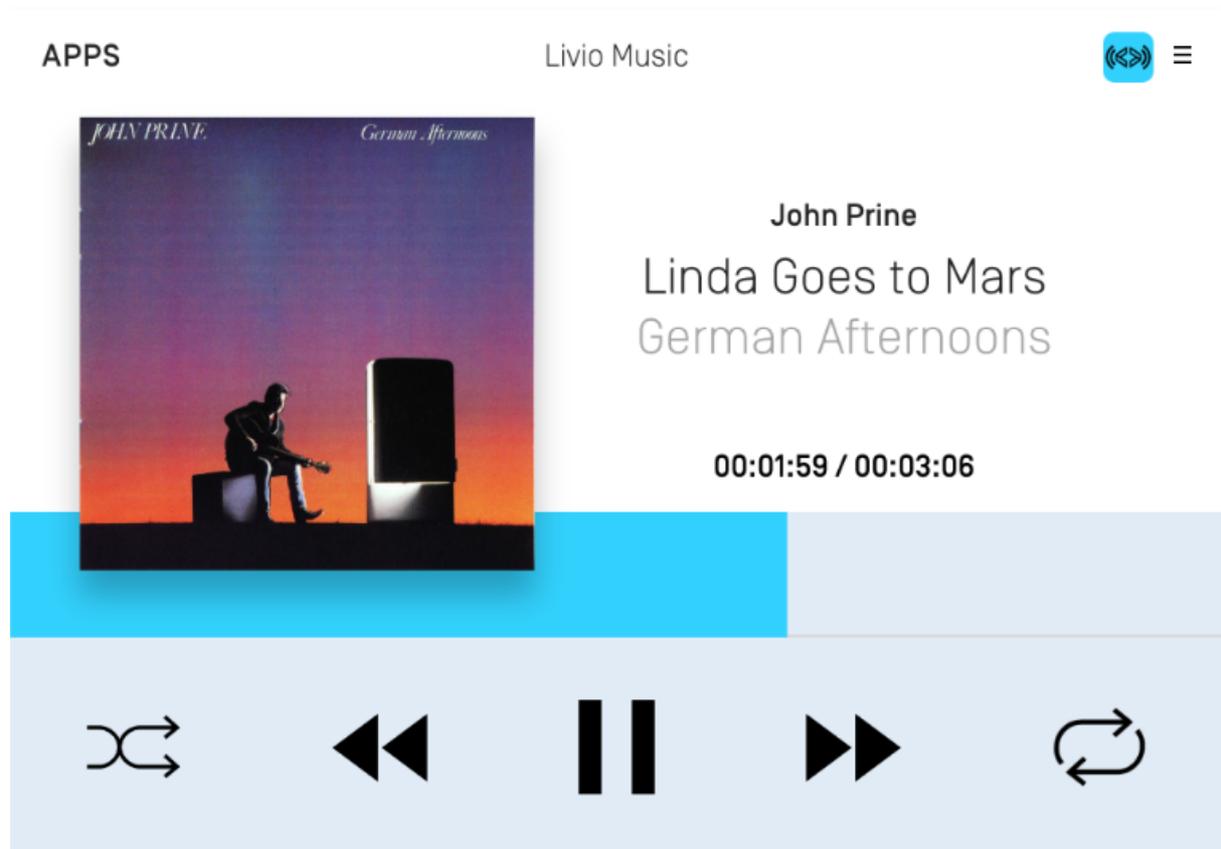
## Subscribing to System Buttons

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	media template only	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

## Subscribe Buttons HMI

In the screenshot below, the pause, seek left and seek right icons are subscribe buttons.



### 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 `appType` to `MEDIA`.

## NOTE

Before library v.6.1 and SDL Core v.5.0, `Ok` and `PlayPause` were combined into `Ok`. Subscribing to `Ok` will, in v.6.1, also subscribe you to `PlayPause`. This means that for the time being, *you should not simultaneously subscribe to `Ok` and `PlayPause`*. In a future major version, this will change. For now, only subscribe to either `Ok` or `PlayPause` and the library will execute the right action based on the version of Core to which you are connected.

## OBJECTIVE-C

```
SDLSubscribeButton *subscribeButton = [[SDLSubscribeButton alloc]
initWithButtonName:SDLButtonNamePlayPause handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
 _Nullable buttonEvent) {
    <#subscribe button selected#>
}];
[manager sendRequest:subscribeButton withResponseHandler:^(
    __typeof SDLRPCRequest * _Nullable request, __typeof 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#>
}
```

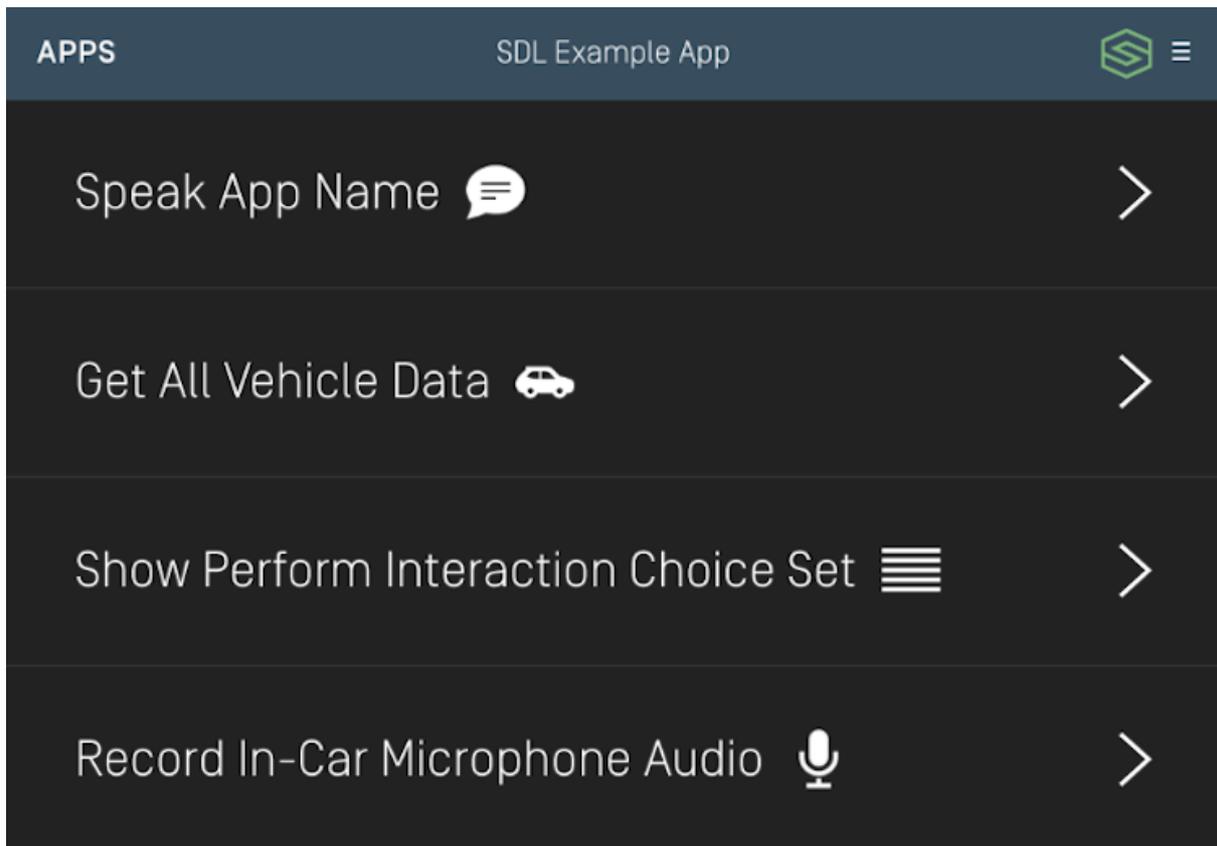
# Main Menu

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.

---

## MENU TEMPLATE



### NOTE

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

# Adding Menu Items

The best way to create and update your menu is to 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: <#NSString#>
icon: <#SDLArtwork#> voiceCommands: <#@[NSString]#> 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#>
}

sdIManager.screenManager.menu = [cell]
```

## Adding Submenus

Adding a submenu is as simple as adding subcells to a `SDLMenuCell`. The submenu is automatically displayed when selected by the user. Currently menus only support one layer of subcells.

## OBJECTIVE-C

```
// Create the inner menu cell
SDLMenuCell *cell = [[SDLMenuCell alloc] initWithTitle: <#NSString#>
icon: <#SDLArtwork#> voiceCommands: <#@[NSString]#> 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:<#
NSString#> icon:<#SDLArtwork?#> subCells:@[cell]];
self.sdIManager.screenManager.menu = @[submenuCell];
```

## SWIFT

```
// Create the inner 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#>
}

let submenuCell = SDLMenuCell(title: <#String#>, icon: <#
SDLArtwork?#>, subCells:[cell])
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 and Changing Menu Items

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

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

# Using RPCs

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

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

## Popup Menus and Keyboards

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

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

selection of options to your user. They can also be used to show a keyboard that lets your user perform a search or provide user input.

## Presenting a Popup Menu

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

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

## Creating Cells

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

## NOTE

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

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

## Preloading Cells

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

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

## OBJECTIVE-C

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

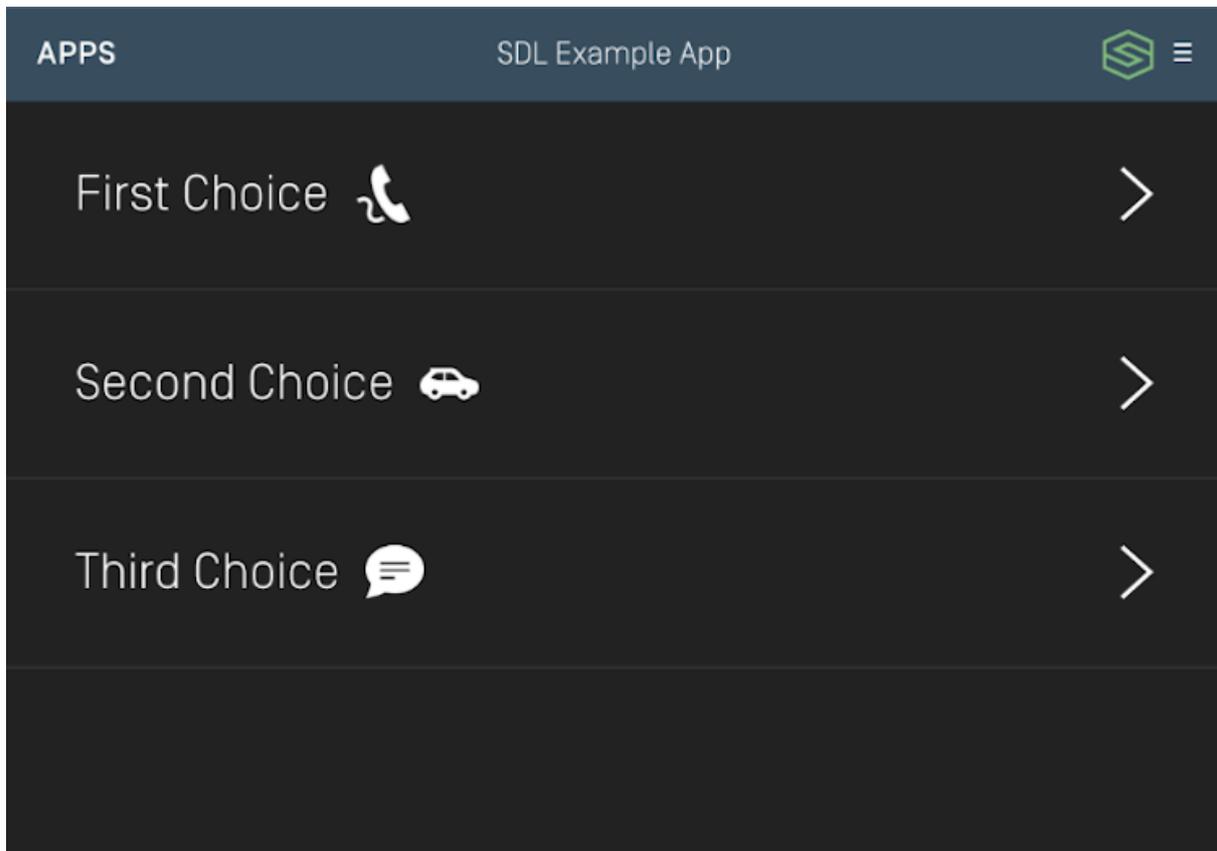
## SWIFT

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

## Presenting a Menu

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

## MENU - LIST



## MENU - ICON

### SYNC 3 - Icon Only Interaction Layout

#### 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 `SDLChoiceCells` into an `SDLChoiceSet`.

### NOTE

If the `SDLChoiceSet` contains an invalid set of `SDLChoiceCells`, 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:<#
(nullable NSArray<SDLChoiceCell *> *)#>];
```

## SWIFT

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

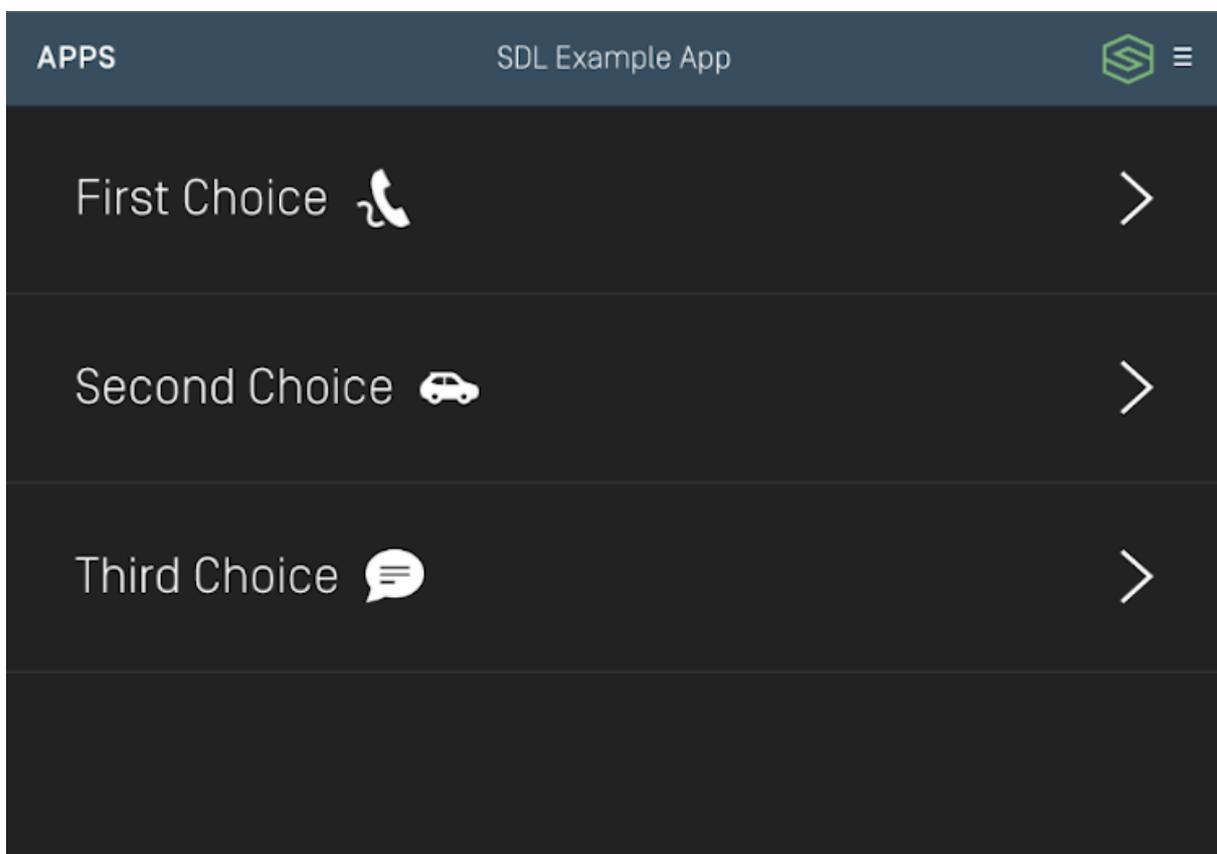
Finally, you will present the menu. When you do so, you must choose a `mode` to present it in. If you have no `vrCommands` on the choice cell you should choose `manualOnly`. If `vrCommands` are available, you may choose `voiceRecognitionOnly` or `both`.

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

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

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

## MENU - MANUAL ONLY MODE



## MENU - VOICE ONLY MODE

SYNC 3 - Menu - Voice Only

## OBJECTIVE-C

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

## SWIFT

```
sdlManager.screenManager.present(<#choiceSet: SDLChoiceSet#>,  
mode: <#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.

### MENU WITH SEARCH

SYNC 3 - Menu with Search Interaction Layout

## OBJECTIVE-C

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

## SWIFT

```
sdIManager.screenManager.presentSearchableChoiceSet(<#choiceSet:
SDLChoiceSet#>, mode: <#SDLInteractionMode#>, with: <#
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.sdIManager.screenManager deleteChoices:<#(nonnull NSArray<
SDLChoiceCell *> *)#>];
```

## SWIFT

```
sdIManager.screenManager.deleteChoices(<#choices: [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.

## KEYBOARD SEARCH

SYNC 3 - Keyboard Search

## OBJECTIVE-C

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

## SWIFT

```
sdlManager.screenManager.presentKeyboard(withInitialText: <#String#>, delegate: <#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#>
}
```

SWIFT

```

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 `Choice`, `CreateInteractionChoiceSet`, and `PerformInteraction` RPC requests. You will need to create `Choice`s, bundle them into `CreateInteractionChoiceSet`s, and then present those choice sets via a `PerformInteraction` request. As this is no longer a recommended course of action, we will leave it to you to figure out how to manually do it.

## Alerts

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

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.

### 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 Layouts

## Alert With No Soft Buttons



### NOTE

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

## Alert With Soft Buttons



## Creating the Alert

### Text

#### OBJECTIVE-C

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

## SWIFT

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

## Buttons

### OBJECTIVE-C

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

SDLSoftButton *button1 = [[SDLSoftButton alloc] initWithType:
SDLSoftButtonTypeText text:@"<#Button Text#>" image:nil highlighted:
false buttonId:<#Soft Button Id#> systemAction:
SDLSystemActionDefaultAction handler:^(SDLOnButtonPress *_Nullable
buttonPress, SDLOnButtonEvent *_Nullable buttonEvent) {
    if (buttonPress == nil) {
        return;
    }
    <#Button has been pressed#>
}];

SDLSoftButton *button2 = [[SDLSoftButton alloc] initWithType:
SDLSoftButtonTypeText text:<#Button Text#> image:nil highlighted:
false buttonId:<#Soft Button Id#> systemAction:
SDLSystemActionDefaultAction handler:^(SDLOnButtonPress *_Nullable
buttonPress, SDLOnButtonEvent *_Nullable buttonEvent) {
    if (buttonPress == nil) {
        return;
    }
    <#Button has been pressed#>
}];

alert.softButtons = @[button1, button2];
```

## SWIFT

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

let button1 = SDLSoftButton(type: .text, text: <#Button Text#>, image : nil, highlighted: false, buttonId: <#Soft Button Id#>, systemAction: . defaultAction, handler: { buttonPress, buttonEvent in
    guard buttonPress != nil else { return }
    <#Button has been pressed#>
})

let button2 = SDLSoftButton(type: .text, text: <#Button Text#>, image : nil, highlighted: false, buttonId: <#Soft Button Id#>, systemAction: . defaultAction, handler: { buttonPress, buttonEvent in
    guard buttonPress != nil else { return }
    <#Button has been pressed#>
})

alert.softButtons = [button1, button2]
```

## Timeouts

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

## OBJECTIVE-C

```
// Duration timeout is in milliseconds
alert.duration = @(4000);
```

## SWIFT

```
// Duration timeout is in milliseconds  
alert.duration = 4000 as NSNumber
```

## Progress Indicator

Not all OEMs support a progress indicator. If supported, the alert will show an animation that indicates that the user must wait (e.g. a spinning wheel or hourglass, etc). If omitted, no progress indicator will be shown.

## OBJECTIVE-C

```
alert.progressIndicator = @YES;
```

## SWIFT

```
alert.progressIndicator = true as NSNumber
```

## Text-To-Speech

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

---

## TEXT

## OBJECTIVE-C

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

## SWIFT

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

---

## SOUND FILE

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

## OBJECTIVE-C

```
alert.ttsChunks = [SDLTTSChunk fileChunksWithName:@"<#Name#>"  
];
```

## SWIFT

```
alert.ttsChunks = SDLTTSChunk.fileChunks(withName: "<#Name#>")
```

## Play Tone

To play the alert tone when the alert appears and before the text-to-speech is spoken, set `playTone` to `true`.

### OBJECTIVE-C

```
alert.playTone = @YES;
```

### SWIFT

```
alert.playTone = true as NSNumber
```

## Showing the Alert

### OBJECTIVE-C

```
[self.sdlManager sendRequest:alert withResponseHandler:^(
    SDLRPCRequest *request, SDLRPCResponse *response, NSError *error)
{
    if (![response.resultCode isEqualToEnum:SDLResultSuccess]) { return
; }
    <#alert was dismissed successfully#>
}];
```

## SWIFT

```
sdIManager.send(request: alert) { (request, response, error) in
    guard response?.resultCode == .success else { return }
    <#alert was dismissed successfully#>
}
```

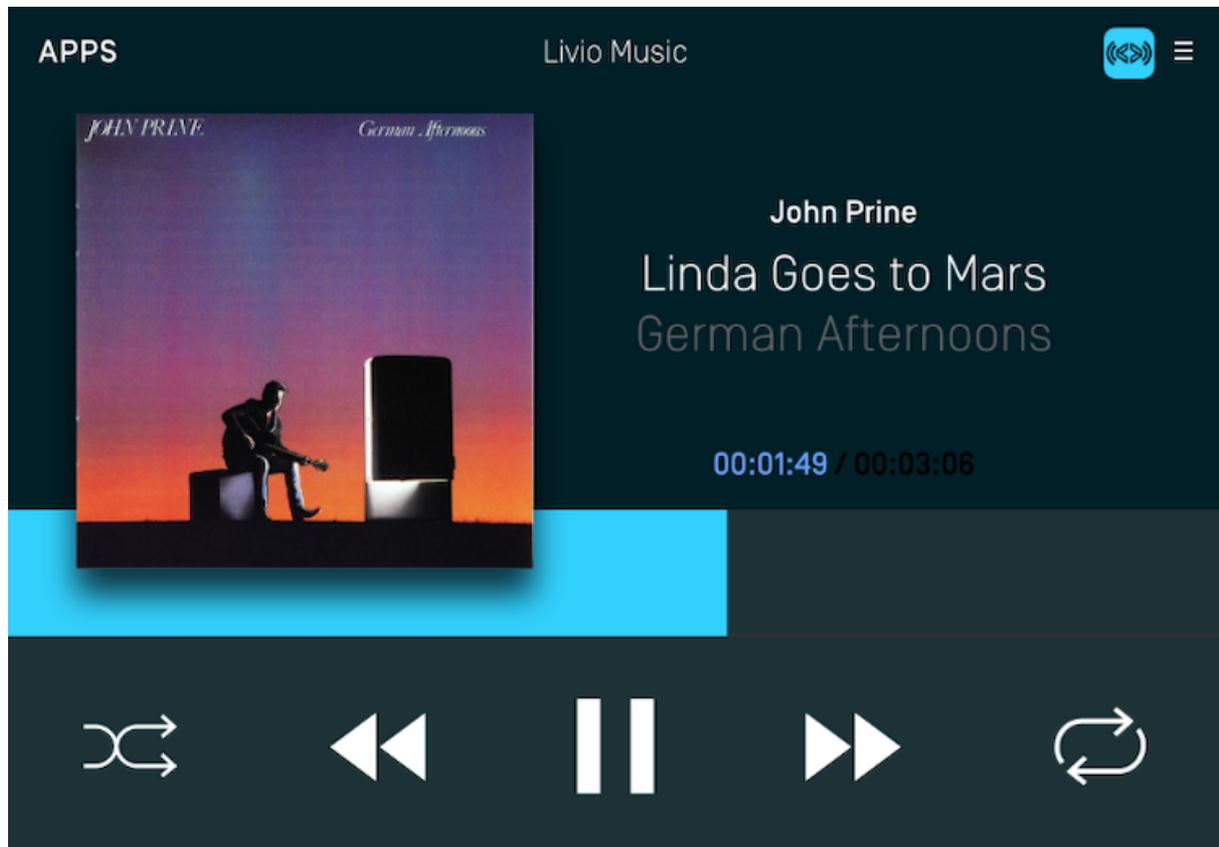
# Media Clock

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.

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

### NOTE

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



## Counting Up

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

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

## OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [SDLSetMediaClockTimer
countUpFromStartTimeInterval:30 toEndTimeInterval:253
playPauseIndicator:SDLAudioStreamingIndicatorPause];
[self.sdlManager sendRequest:mediaClock];
```

## SWIFT

```
let mediaClock = SDLSetMediaClockTimer.countUp(from: 30, to: 253,
playPauseIndicator: .pause)
sdlManager.send(mediaClock)
```

# Counting Down

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

## OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [SDLSetMediaClockTimer
countDownFromStartTimeInterval:600 toEndTimeInterval:0
playPauseIndicator:SDLAudioStreamingIndicatorPause];
[self.sdlManager sendRequest:mediaClock];
```

## SWIFT

```
let mediaClock = SDLSetMediaClockTimer.countDown(from: 600, to: 0,  
playPauseIndicator: .pause)  
sdIManager.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  
pauseWithPlayPauseIndicator:SDLAudioStreamingIndicatorPlay];  
[self.sdIManager sendRequest:mediaClock];
```

```
SDLSetMediaClockTimer *mediaClock = [SDLSetMediaClockTimer  
resumeWithPlayPauseIndicator:SDLAudioStreamingIndicatorPause];  
[self.sdIManager sendRequest:mediaClock];
```

```
SDLSetMediaClockTimer *mediaClock = [SDLSetMediaClockTimer  
updatePauseWithNewStartTimeInterval:60 endTimeInterval:240  
playPauseIndicator:SDLAudioStreamingIndicatorPlay];  
[self.sdlManager sendRequest:mediaClock];
```

## SWIFT

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

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

```
let mediaClock = SDLSetMediaClockTimer.pause(newStart: 60, newEnd:  
240, playPauseIndicator: .play)  
sdlManager.send(mediaClock)
```

## Clearing the Timer

Clearing the timer removes it from the screen.

## OBJECTIVE-C

```
SDLSetMediaClockTimer *mediaClock = [SDLSetMediaClockTimer  
clearWithPlayPauseIndicator:SDLAudioStreamingIndicatorPlay];  
[self.sdlManager sendRequest:mediaClock];
```

## SWIFT

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

# Updating the Audio Indicator

The audio indicator is, essentially, the play / pause button. As of library v.6.1, 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, if audio is currently playing you can update the the play/pause button to show the pause icon. On older head units, the audio indicator shows an icon with both the play and pause indicators and the icon can not be updated.

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

# Slider

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

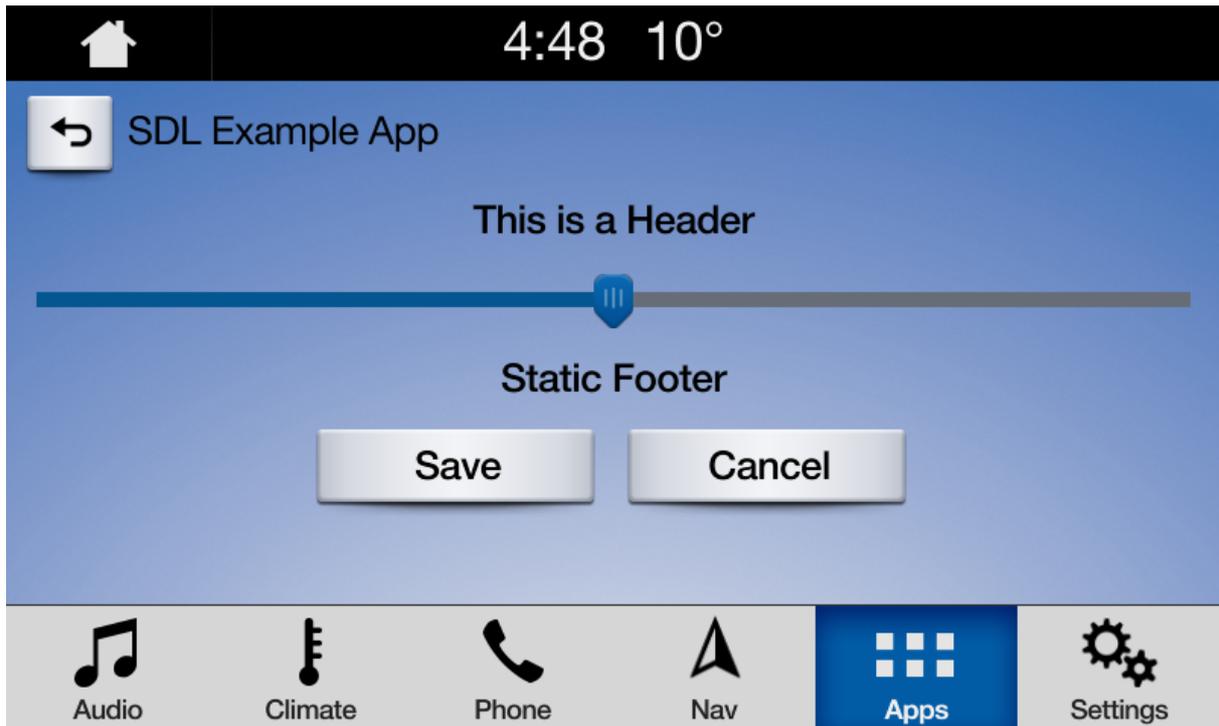
## NOTE

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

## Slider with Static Footer

A slider popup with a static footer displays a single, optional, footer message below the slider UI.

# Slider UI



## Creating the Slider

## OBJECTIVE-C

```
// Create a slider with number of ticks, starting position 'tick number', a
header message, an optional footer message, and a timeout of 30
seconds
SDLSlider *sdSlider = [[SDLSlider alloc] initWithNumTicks:5 position:1
sliderHeader:@"This is a Header" sliderFooter:@"Static Footer" timeout:
30000];

// Send the slider RPC request with handler
[manager sendRequest:sdSlider withResponseHandler:^(__kindof
SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse *
_Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error getting the SDLSlider response");
        return;
    }
}

// Create a SDLSlider response object from the handler response
SDLSliderResponse *sdSliderResponse = (SDLSliderResponse *)
response;
NSInteger position = sdSliderResponse.sliderPosition.
unsignedIntegerValue;

<#Use the slider position#>
}];
```

## SWIFT

```
// Create a slider with number of ticks, starting position 'tick number', a
header message, an optional footer message, and a timeout of 30
seconds
let slider = SDLSlider(numTicks: 5, position: 1, sliderHeader: "This is a
Header", sliderFooter: "Static Footer", timeout: 30000)

// Send the slider RPC request with handler
manager.send(request: slider, responseHandler: { (req, res, err) in
    // Create a SDLSlider response object from the handler response
    guard let response = res as? SDLSliderResponse, response.
resultCode == .success, let position = response.sliderPosition.intValue
else { return }

    <#Use the slider position#>
})
```

# Slider with Dynamic Footer

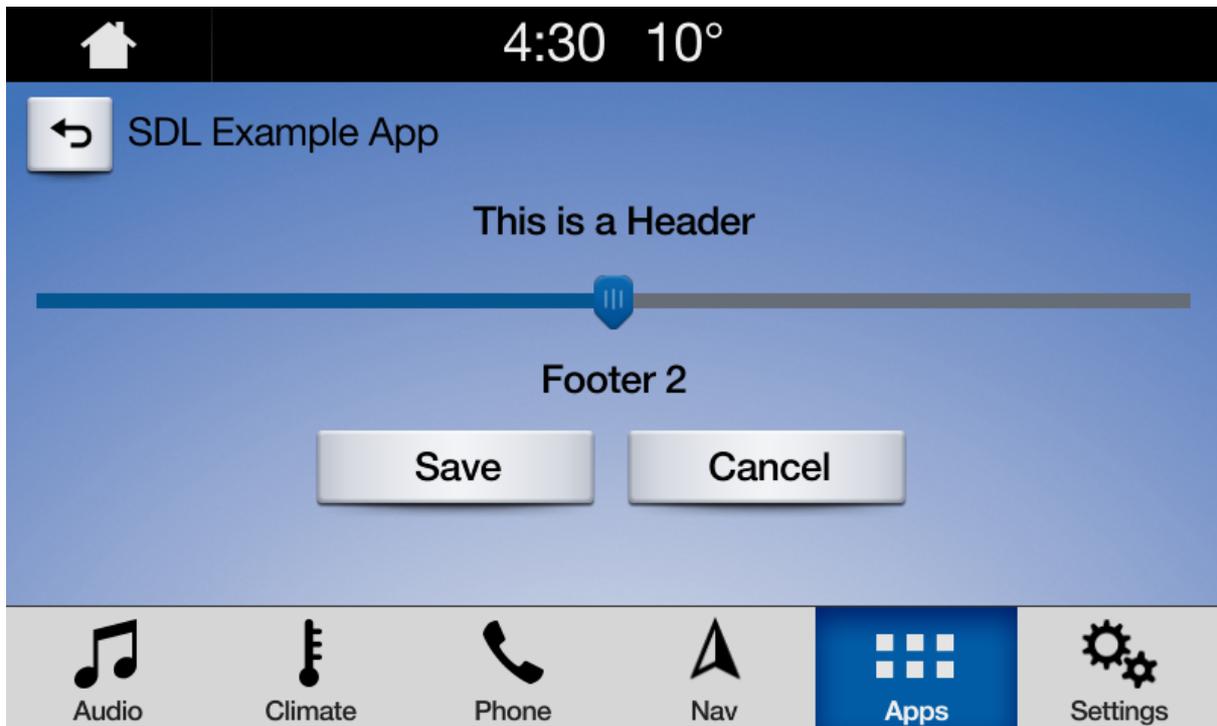
This type of slider will have a different footer message displayed for each position of the slider. The footer is an optional parameter. The footer message displayed will be based off of the slider's current position. The footer array should be the same length as `numTicks` because each footer must correspond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

## Slider UI

DYNAMIC SLIDER IN POSITION 1



## DYNAMIC SLIDER IN POSITION 2



## Creating the Slider

## OBJECTIVE-C

```
// Create an array of footers to display to the user. This array *must* be
equal in length to the number of ticks of the slider you create below.
NSArray<NSString*> *footers = @[@"Footer 1", @"Footer 2",
@"Footer 3"];

// Create a slider with number of ticks, starting position 'tick number', a
header message, and an optional footer array, and a timeout of 30
seconds
SDLSlider *sdSlider = [[SDLSlider alloc] initWithNumTicks:3 position:1
sliderHeader:@"This is a Header" sliderFooters:footers timeout:30000];

// Send the slider RPC request with handler
[manager sendRequest:sdSlider withResponseHandler:^(__kindof
SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse *
_Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error getting the SDLSlider response");
        return;
    }
}

// Create a SDLSlider response object from the handler response
SDLSliderResponse *sdSliderResponse = (SDLSliderResponse *)
response;
NSInteger position = sdSliderResponse.sliderPosition.
unsignedIntegerValue;

<#Use the slider position#>
}];
```

## SWIFT

```
// Create an array of footers to display to the user. This array *must* be
equal in length to the number of ticks of the slider you create below.
let footers = ["Footer 1", "Footer 2", "Footer 3"]

// Create a slider with number of ticks, starting position 'tick number', a
header message, and an optional footer array, and a timeout of 30
seconds
let slider = SDLSlider(numTicks: 3, position: 1, sliderHeader: "This is a
Header", sliderFooters: footers, timeout: 30000)
manager.send(request: slider, responseHandler: { (req, res, err) in
    // Create a SDLSlider response object from the handler response
    guard let response = res as? SDLSliderResponse, response.
resultCode == .success, let position = response.sliderPosition.intValue
else { return }

    <#Use the slider position#>
})
```

# Scrollable Message

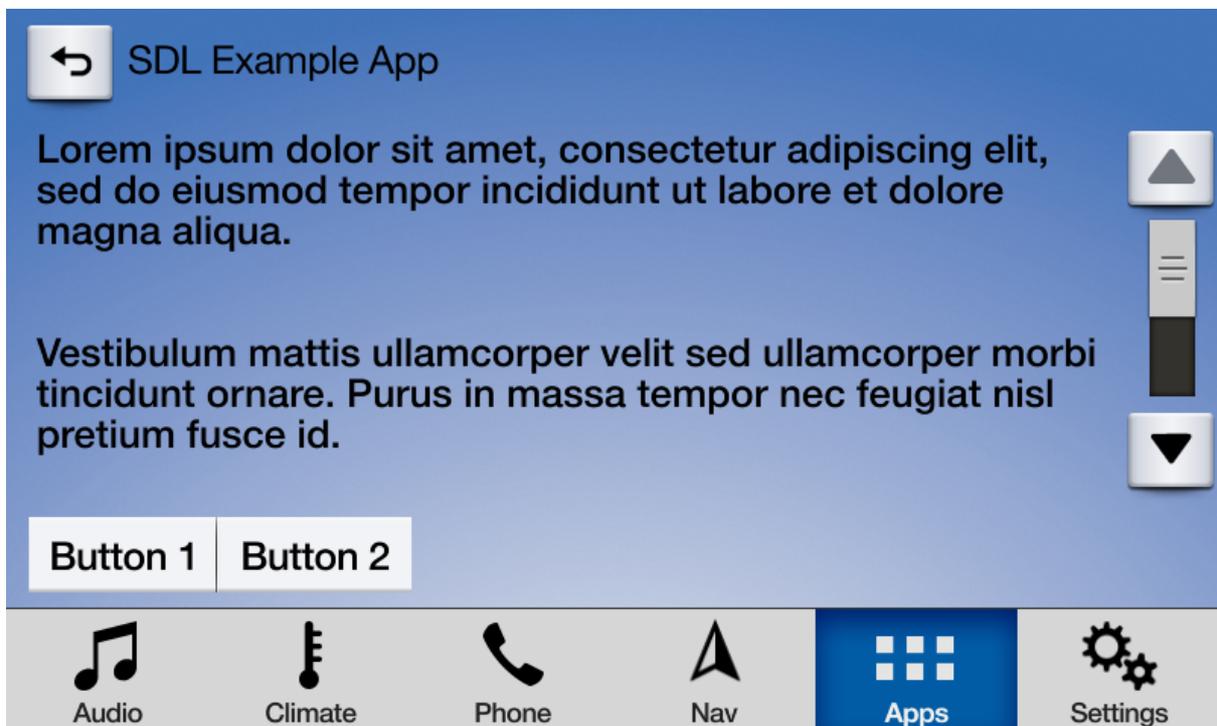
A `SDLScrollableMessage` creates an overlay containing a large block of formatted text that can be scrolled. `SDLScrollableMessage` contains a body of text, a message timeout, and up to 8 soft buttons depending on head unit. You must check the `DisplayCapabilities` to get the max number of `SoftButtons` allowed by the head unit for a `ScrollableMessage`.

You simply create an `SDLScrollableMessage` RPC request and send it to display the Scrollable Message.

## NOTE

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

# Scrollable Message UI



## Creating the Scrollable Message

## OBJECTIVE-C

```
// Create SoftButton Array
NSMutableArray<SDLSoftButton *> *softButtons = [[NSMutableArray
alloc] init];

// Create Message To Display
NSString *scrollableMessageString = [NSString stringWithFormat:
@"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.\n\n
Vestibulum mattis ullamcorper velit sed ullamcorper morbi tincidunt
ornare. Purus in massa tempor nec feugiat nisl pretium fusce id.\n\n
Pharetra convallis posuere morbi leo urna molestie at elementum eu.
Dictum sit amet justo donec enim diam."];

// Create a timeout of 50 seconds
UInt16 scrollableMessageTimeout = 50000;

// Create SoftButtons
SDLSoftButton *scrollableSoftButton = [[SDLSoftButton alloc]
initWithType:SDLSoftButtonTypeText text:@"Button 1" image:nil
highlighted:NO buttonId:111 systemAction:nil handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_Nullable buttonEvent) {
    if (buttonPress == nil) { return;
    // Create a custom action for the selected button
}];
SDLSoftButton *scrollableSoftButton2 = [[SDLSoftButton alloc]
initWithType:SDLSoftButtonTypeText text:@"Button 2" image:nil
highlighted:NO buttonId:222 systemAction:nil handler:^(
SDLOnButtonPress * _Nullable buttonPress, SDLOnButtonEvent *
_Nullable buttonEvent) {
    if (buttonPress == nil) { return; }
    // Create a custom action for the selected button
}];

[softButtons addObject:scrollableSoftButton];
[softButtons addObject:scrollableSoftButton2];

// Create SDLScrollableMessage Object
SDLScrollableMessage *scrollableMessage = [[SDLScrollableMessage
alloc] initWithMessage:scrollableMessageString timeout:
scrollableMessageTimeout softButtons:[softButtons copy]];

// Send the scrollable message
[sdlManager sendRequest:scrollableMessage];
```

## SWIFT

```
// Create SoftButton Array
var softButtons = [SDLSoftButton]()

// Create Message To Display
let scrollableMessageText = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua.

Vestibulum mattis ullamcorper velit sed ullamcorper morbi tincidunt
ornare. Purus in massa tempor nec feugiat nisl pretium fusce id.

Pharetra convallis posuere morbi leo urna molestie at elementum eu.
Dictum sit amet justo donec enim diam.
"""

// Create a timeout of 50 seconds
let scrollableTimeout: UInt16 = 50000

// Create SoftButtons
let scrollableSoftButton = SDLSoftButton(type: .text, text: "Button 1",
image: nil, highlighted: false, buttonId: 111, systemAction: .
defaultAction, handler: { (buttonPress, buttonEvent) in
    guard let press = buttonPress else { return }

    // Create a custom action for the selected button
})
let scrollableSoftButton2 = SDLSoftButton(type: .text, text: "Button 2",
image: nil, highlighted: false, buttonId: 222, systemAction: .
defaultAction, handler: { (buttonPress, buttonEvent) in
    guard let press = buttonPress else { return }

    // Create a custom action for the selected button
})

softButtons.append(scrollableSoftButton)
softButtons.append(scrollableSoftButton2)

// Create SDLScrollableMessage Object
let scrollableMessage = SDLScrollableMessage(message:
scrollableMessageText, timeout: scrollableTimeout, softButtons:
softButtons)

// Send the scrollable message
sdlManager.send(scrollableMessage)
```

# Playing Spoken Feedback

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

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

## Creating the Speak Request

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

## Getting Head Unit Speech Capabilities

To get the head unit's supported speech capabilities, check the `SDLSystemCapabilityManager.speechCapabilities` after successfully connecting to the head unit. Below is a list of commonly supported speech capabilities.

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

## Text Phrase

### OBJECTIVE-C

```
SDLSpeak *speak = [[SDLSpeak alloc] initWithTTS:@"hello"];
```

### SWIFT

```
let speech = SDLSpeak(tts: "hello")
```

## SAPI Phonemes Phrase

### OBJECTIVE-C

```
NSArray<SDLTTChunk *> *sapiPhonemesTTSChecks = [SDLTTChunk  
sapiChunksFromString:@"h eh - l ow 1"];  
SDLSpeak *speak = [[SDLSpeak alloc] initWithTTSChecks:  
sapiPhonemesTTSChecks];
```

## SWIFT

```
let sapiPhonemesTTSChecks = SDLTTSCheck.sapiChecks(from: "h eh -  
low l")  
let speech = SDLSpeak(ttsCheck: sapiPhonemesTTSCheck)
```

# Sending the Speak Request

## OBJECTIVE-C

```
[self.sdlManager sendRequest:speak withResponseHandler:^(__kindof  
SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse *  
_Nullable response, NSError * _Nullable error) {  
    if (![response.resultCode isEqualToEnum:SDLResultSuccess]) {  
        if ([response.resultCode isEqualToEnum:SDLResultDisallowed]) {  
            <#The app does not have permission to use the speech request  
#>  
        } else if ([response.resultCode isEqualToEnum:SDLResultRejected])  
        {  
            <#The request was rejected because a higher priority request  
is in progress#>  
        } else if ([response.resultCode isEqualToEnum:SDLResultAborted])  
        {  
            <#The request was aborted by another higher priority request#  
>  
        } else {  
            <#Some other error occurred#>  
        }  
        return;  
    }  
    <#Speech was successfully spoken#>  
}];
```

## SWIFT

```
sdIManager.send(request: speech) { (request, response, error) in
  guard let response = response as? SDLSpeakResponse else { return
  }
  guard response.resultCode == .success else {
    switch response.resultCode {
    case .disallowed:
      <#The app does not have permission to use the speech request
#>
    case .rejected:
      <#The request was rejected because a higher priority request
is in progress#>
    case .aborted:
      <#The request was aborted by another higher priority request#
>
    default:
      <#Some other error occurred#>
    }
    return
  }
  <#Speech was successfully spoken#>
}
```

# Playing Audio Indications

As of library v.6.1 and SDL Core v.5.0+, you can pass an uploaded audio file's name to `SDLTTSChunk`, allowing any API that takes a text-to-speech parameter to pass and play your audio file. A sports app, for example, could

play a distinctive audio chime to notify the user of a score update alongside an `Alert` request.

## Uploading the Audio File

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

### OBJECTIVE-C

```
SDLFile *audioFile = [[SDLFile alloc] initWithFileURL:<#File location on
disk#> name:<#Audio file name#> persistent:<#True if the file will be
used 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 name#>, persistent: <#True if the file will be used 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 [Uploading Files guide](#).

# Using the Audio File

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

## OBJECTIVE-C

```
SDLAlert *alert = [[SDLAlert alloc] initWithAlertText1:<#nullable  
NSString *#> alertText2:<#nullable NSString *#> duration:<#UInt16#  
>];  
alert.ttsChunks = [SDLTTSChunk fileChunksWithName:<#Audio file  
name#>];  
[self.sdlManager sendRequest:alert];
```

## SWIFT

```
let alert = SDLAlert(alertText1: <#String?#>, alertText2: <#String?#>,  
duration: <#UInt16#>)  
alert.ttsChunks = SDLTTSChunk.fileChunks(withName: <#Audio file  
name#>)  
sdlManager.send(alert)
```

# Setting Up Voice Commands

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

## NOTE

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

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

## Creating Voice Commands

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

## OBJECTIVE-C

```
SDLVoiceCommand *voiceCommand = [[SDLVoiceCommand alloc]
initWithVoiceCommands:@[<#NSString#>] handler:^(
    <#Voice command selected#>
)];

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

## SWIFT

```
let voiceCommand = SDLVoiceCommand(voiceCommands: [<#String#
>]) {
    <#Voice command triggered#>
}

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.

# Getting Microphone Audio

Capturing in-car audio allows developers to interact with users by requesting raw audio data provided to them from the car's microphones. In order to gather

the raw audio from the vehicle, you must leverage the

`SDLPerformAudioPassThru` RPC.

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

#### NOTE

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

## Starting Audio Capture

To initiate audio capture, first construct a `SDLPerformAudioPassThru` request. You must use a sampling rate, bit rate, and audio type supported by the head unit. To get the head unit's supported audio capture capabilities, check the `SDLSystemCapabilityManager.audioPassThruCapabilities` after successfully connecting to the head unit.

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

## 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)
```



## 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

```
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

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

Audio capture can be ended 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 handle the audio pass thru 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 ignore the 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 handle the 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. You will receive a `resultCode` of `SUCCESS`, and should handle the audio pass thru as though it was successful.

## OBJECTIVE-C

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

## SWIFT

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

# Handling the Response

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

## OBJECTIVE-C

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

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

    <#Process audio data#>
}];
```

## SWIFT

```
sdlManager.send(request: audioPassThru) { (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#>
}
```

# Batch Sending RPCs

There are two ways to send multiple requests to the head unit: concurrently and sequentially. Which method you should use depends on the type of RPCs being sent. Concurrently sent requests might finish in a random order and should only be used when none of the requests in the group depend on the response of another, such as when subscribing to several hard buttons. Sequentially sent requests only send the next request in the group when a response has been received for the previously sent RPC. Requests should be sent sequentially when you need to know the result of a previous request before sending the next, like when sending the several different requests needed to create a menu.

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

## Sending Concurrent Requests

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

## OBJECTIVE-C

```
SDLSubscribeButton *subscribeButtonLeft = [[SDLSubscribeButton alloc
] initWithName:SDLButtonNameSeekLeft];
SDLSubscribeButton *subscribeButtonRight = [[SDLSubscribeButton
alloc] initWithName:SDLButtonNameSeekRight];
[self.sdlManager sendRequests:@[subscribeButtonLeft,
subscribeButtonRight] completionHandler:^(__typeof(SDLRPCRequest *
_Nonnull request, __typeof(SDLRPCResponse * _Nullable response,
NSError * _Nullable error, float percentComplete) {
    <#Called as each request completes#>
} completionHandler:^(BOOL success) {
    <#Called when all requests complete#>
}]);
```

## SWIFT

```
let subscribeButtonLeft = SDLSubscribeButton(name:
SDLButtonNameSeekLeft)
let subscribeButtonRight = SDLSubscribeButton(name:
SDLButtonNameSeekRight)
sdlManager.send([subscribeButtonLeft, subscribeButtonRight],
progressHandler: { (request, response, error, percentComplete) in
    <#Called as each request completes#>
}) { success in
    <#Called when all requests complete#>
}
```

# 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] completionHandler:^(
BOOL(__kindof SDLRPCRequest * _Nonnull request, __kindof
SDLRPCResponse * _Nullable response, NSError * _Nullable error, float
percentComplete) {
    <#Called as each request completes#>
    return YES;
} completionHandler:^(BOOL success) {
    <#Called when all requests complete#>
}];
```

## 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], completionHandler: { (request, response, error,
percentageCompleted) -> Bool in
    <#Called as each request completes#>
    return true
}) { success in
    <#Called when all requests complete#>
}
```

# Retrieving Vehicle Data

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

## NOTE

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

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
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.
Cloud App Vehicle Id	cloudAppVehicleID	The id for the vehicle when connecting to cloud applications 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
Cluster Mode Status	clusterModeStatus	

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

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

## OBJECTIVE-C

```
SDLGetVehicleData *getGPSData = [[SDLGetVehicleData alloc] init];
getGPSData.gps = @YES;
[self.sdlManager sendRequest:getGPSData withResponseHandler:^(
    __typeof SDLRPCRequest * _Nullable request, __typeof SDLRPCResponse
    * _Nullable response, NSError * _Nullable error) {
    SDLGetVehicleDataResponse *vehicleDataResponse = (
    SDLGetVehicleDataResponse *)response;
    SDLResult resultCode = vehicleDataResponse.resultCode;

    if (![resultCode isEqualToEnum:SDLResultSuccess]) {
        if ([resultCode isEqualToEnum:SDLResultDisallowed]) {
            <#The app does not have permission to access this vehicle
            data#>
        } else if ([resultCode isEqualToEnum:SDLResultRejected]) {
            <#The app does not have permission to access this vehicle
            data because of the app state (i.e. app is closed or in background)#>
        } else if ([resultCode isEqualToEnum:
    SDLResultVehicleDataNotAllowed]) {
            <#The data is not available or responding on the system#>
        } else if ([resultCode isEqualToEnum:
    SDLResultVehicleDataNotAvailable]) {
            <#The user has turned off access to vehicle data, and it is
            globally unavailable to mobile applications#>
        } else {
            <#Some other error occurred#>
        }
        return;
    }

    SDLGPSData *gpsData = vehicleDataResponse.gps;
    if (gpsData == nil) { return; }
    <#Use the GPS data#>
}];
```

## SWIFT

```
let getGPSData = SDLGetVehicleData()
getGPSData.gps = true as NSNumber
sdManager.send(request: getGPSData) { (request, response, error) in
    guard let response = response as? SDLGetVehicleDataResponse else
    { return }
    guard response.resultCode == .success else {
        switch response.resultCode {
            case .disallowed:
                <#The app does not have permission to access this vehicle
data#>
            case .rejected:
                <#The app does not have permission to access this vehicle
data because of the app state (i.e. app is closed or in background)#>
            case .vehicleDataNotAllowed:
                <#The data is not available or responding on the system#>
            case .vehicleDataNotAvailable:
                <#The user has turned off access to vehicle data, and it is
globally unavailable to mobile applications#>
            default:
                <#Some other error occurred#>
        }
    }
    return
}

guard let gpsData = response.gps else { return }
<#Use the GPS data#>
}
```

# Subscribing to Vehicle Data

Subscribing to vehicle data allows you to get notifications whenever new data is available. You should not rely upon getting this data in a consistent manner. New vehicle data is available roughly every second, but this is totally dependent on which head unit you are connected to.

**First**, register to observe the `SDLDidReceiveVehicleDataNotification` notification:

## OBJECTIVE-C

```
// sdl_ios v6.3+
[self.sdlManager subscribeToRPC:SDLDidReceiveVehicleDataNotification
withObserver:self selector:@selector(vehicleDataAvailable:)];

// Pre sdl_ios v6.3
[[NSNotificationCenter defaultCenter] addObserver:self selector:
@selector(vehicleDataAvailable:) name:
SDLDidReceiveVehicleDataNotification object:nil];
```

## SWIFT

```
// sdl_ios v6.3+
sdlManager.subscribe(to: .SDLDidReceiveVehicleData, observer: self,
selector: #selector(vehicleDataAvailable(_:)))

// Pre sdl_ios v6.3
NotificationCenter.default.addObserver(self, selector: #selector(
vehicleDataAvailable(_:)), name: .SDLDidReceiveVehicleData, object: nil
)
```

**Second**, send the `SubscribeVehicleData` request:

## OBJECTIVE-C

```
SDLSubscribeVehicleData *subscribeGPSData = [[
SDLSubscribeVehicleData alloc] init];
subscribeGPSData.gps = @YES;

[self.sdlManager sendRequest:subscribeGPSData withResponseHandler:
^(__typeof SDLRPCRequest * _Nullable request, __typeof
SDLRPCResponse * _Nullable response, NSError * _Nullable error) {
    SDLSubscribeVehicleDataResponse *vehicleDataResponse = (
SDLSubscribeVehicleDataResponse *)response;
    if (![vehicleDataResponse.resultCode isEqualToEnum:
SDLResultSuccess]) {
        if ([vehicleDataResponse.resultCode isEqualToEnum:
SDLResultDisallowed]) {
            <#The app does not have permission to access this vehicle
data#>
        } else if ([vehicleDataResponse.resultCode isEqualToEnum:
SDLResultUserDisallowed]) {
            <#The user has not granted access to this type of vehicle data
item at this time#>
        } else if ([vehicleDataResponse.resultCode isEqualToEnum:
SDLResultIgnored]) {
            SDLVehicleDataResult *gpsData = vehicleDataResponse.gps;
            if ([gpsData.resultCode isEqualToEnum:
SDLVehicleDataResultCodeDataAlreadySubscribed]) {
                <#Ignoring request because the app is already subscribed to
GPS data#>
            } else if ([gpsData.resultCode isEqualToEnum:
SDLVehicleDataResultCodeVehicleDataNotAvailable]) {
                <#The app has permission to access to the GPS data, but the
vehicle does not provide it#>
            } else {
                <#Request ignored for some other reason#>
            }
        } else {
            <#Some other error occurred#>
        }
        return;
    }

    <#Successfully subscribed to GPS data#>
}];
```

## SWIFT

```
let subscribeGPSData = SDLSubscribeVehicleData()
subscribeGPSData.gps = true as NSNumber

sdIManager.send(request: subscribeGPSData) { (request, response,
error) in
    guard let response = response as?
SDLSubscribeVehicleDataResponse else { return }
    guard response.resultCode == .success else {
        switch response.resultCode {
            case .disallowed:
                <#The app does not have permission to access this vehicle
data#>
            case .userDisallowed:
                <#The user has not granted access to this type of vehicle data
item at this time#>
            case .ignored:
                guard let gpsData = response.gps else { break }
                switch gpsData.resultCode {
                    case .dataAlreadySubscribed:
                        <#Ignoring request because the app is already subscribed to
GPS data#>
                    case .vehicleDataNotAvailable:
                        <#The app has permission to access to the GPS data, but the
vehicle does not provide it#>
                    default:
                        <#Request ignored for some other reason#>
                }
            default:
                <#Some other error occurred#>
        }
        return
    }

    <#Successfully subscribed to GPS data#>
}
```

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

## OBJECTIVE-C

```
- (void)vehicleDataAvailable:(SDLRPCNotificationNotification *)
notification {
    SDLOnVehicleData *onVehicleData = (SDLOnVehicleData *)
notification.notification;
    SDLGPSData *gps = onVehicleData.gps;
    if (gps == nil) { return; }
    <#Use the GPS data#>
}
```

## SWIFT

```
func vehicleDataAvailable(_ notification: SDLRPCNotificationNotification)
{
    guard let onVehicleData = notification.notification as?
SDLOnVehicleData, let gpsData = onVehicleData.gps else {
        return
    }
    <#Use the GPS data#>
}
```

# Unsubscribing from Vehicle Data

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

## OBJECTIVE-C

```
SDLUnsubscribeVehicleData *unsubscribeGPSData = [[
SDLUnsubscribeVehicleData alloc] init];
unsubscribeGPSData.gps = @YES;

[self.sdlManager sendRequest:unsubscribeGPSData
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
__kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
error) {
    SDLUnsubscribeVehicleDataResponse *vehicleDataResponse = (
SDLUnsubscribeVehicleDataResponse*)response;

    if (![vehicleDataResponse.resultCode isEqualToEnum:
SDLResultSuccess]) {
        if ([vehicleDataResponse.resultCode isEqualToEnum:
SDLResultDisallowed]) {
            <#The app does not have permission to access this vehicle
data#>
        } else if ([vehicleDataResponse.resultCode isEqualToEnum:
SDLResultUserDisallowed]) {
            <#The user has not granted access to this type of vehicle data
item at this time#>
        } else if ([vehicleDataResponse.resultCode isEqualToEnum:
SDLResultIgnored]) {
            SDLVehicleDataResult *gpsData = vehicleDataResponse.gps;
            if ([gpsData.resultCode isEqualToEnum:
SDLVehicleDataResultCodeDataNotSubscribed]) {
                <#The app has access to this data item but ignoring since
the app is already unsubscribed to GPS data#>
            } else {
                <#Request ignored for some other reason#>
            }
        } else {
            <#Some other error occurred#>
        }
        return;
    }

    <#Successfully unsubscribed to GPS data#>
}];
```

## SWIFT

```
let unsubscribeGPSData = SDLUnsubscribeVehicleData()
unsubscribeGPSData.gps = true as NSNumber

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

    guard response.resultCode == .success else {
        switch response.resultCode {
        case .disallowed:
            <#The app does not have permission to access this vehicle
data#>
        case .userDisallowed:
            <#The user has not granted access to this type of vehicle data
item at this time#>
        case .ignored:
            guard let gpsData = response.gps else { break }
            switch gpsData.resultCode {
            case .dataNotSubscribed:
                <#The app has access to this data item but ignoring since
the app is already unsubscribed to GPS data#>
            default:
                <#Request ignored for some other reason#>
            }
        default:
            <#Some other error occurred#>
        }
        return
    }

    <#Successfully unsubscribed to GPS data#>
}
```

# Remote Control Vehicle Features

The remote control framework allows apps to control certain modules, such as climate, radio, seat, lights, etc., within a vehicle.

## NOTE

Not all head units support this feature. If using this feature in your app you will most likely need to request permission from the vehicle manufacturer.

## Why Use Remote Control?

Consider the following scenarios:

- A radio application wants to use the in-vehicle radio tuner. It needs the functionality to select the radio band (AM/FM/XM/HD/DAB), tune the radio frequency or change the radio station, as well as obtain general radio information for decision making.
- A climate control application needs to turn on the AC, control the air circulation mode, change the fan speed and set the desired cabin temperature.

- A user profile application wants to remember users' favorite settings and apply it later automatically when the users get into the same/another vehicle.

## Remote Control Modules

Currently, the remote control feature supports these modules:

REMOTE CONTROL MODULES
Climate
Radio
Seat
Audio
Light
HMI Settings

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

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
<b>Climate</b>	Current Cabin Temperature		Get/Notification	read only, value range depends on OEM
	Desired Cabin Temperature		Get/Set/Notification	value range depends on OEM
	AC Setting	on, off	Get/Set/Notification	
	AC MAX Setting	on, off	Get/Set/Notification	
	Air Recirculation Setting	on, off	Get/Set/Notification	
	Auto AC Mode Setting	on, off	Get/Set/Notification	
	Defrost Zone Setting	front, rear, all, none	Get/Set/Notification	
	Dual Mode Setting	on, off	Get/Set/Notification	
	Fan Speed Setting	0%-100%	Get/Set/Notification	
	Ventilation Mode Setting	upper, lower, both, none	Get/Set/Notification	
<b>Radio</b>	Radio Enabled	true,false	Get/Set/Notification	read only, all other radio control items need radio enabled to work
	Radio Band	AM,FM,XM	Get/Set/Notification	

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Radio Frequency		Get/Set/Notification	value range depends on band
	Radio RDS Data		Get/Notification	read only
	Available HD Channel	1-3	Get/Notification	read only
	Current HD Channel	1-3	Get/Set/Notification	
	Radio Signal Strength		Get/Notification	read only
	Signal Change Threshold		Get/Notification	read only
	Radio State	Acquiring, acquired, multicast, not_found	Get/Notification	read only
<b>Seat</b>	Seat Heating Enabled	true, false	Get/Set/Notification	Indicates whether heating is enabled for a seat
	Seat Cooling Enabled	true, false	Get/Set/Notification	Indicates whether cooling is enabled for a seat
	Seat Heating level	0-100%	Get/Set/Notification	Level of the seat heating
	Seat Cooling level	0-100%	Get/Set/Notification	Level of the seat cooling

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Seat Horizontal Position	0-100%	Get/Set/Notification	Adjust a seat forward/backward, 0 means the nearest position to the steering wheel, 100% means the furthest position from the steering wheel
	Seat Vertical Position	0-100%	Get/Set/Notification	Adjust seat height (up or down) in case there is only one actuator for seat height, 0 means the lowest position, 100% means the highest position

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENT S
	Seat-Front Vertical Position	0-100%	Get/Set/ Notification	Adjust seat front height (in case there are two actuators for seat height), 0 means the lowest position, 100% means the highest position
	Seat-Back Vertical Position	0-100%	Get/Set/ Notification	Adjust seat back height (in case there are two actuators for seat height), 0 means the lowest position, 100% means the highest position

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Seat Back Tilt Angle	0-100%	Get/Set/Notification	Backrest recline, 0 means the angle that back top is nearest to the steering wheel, 100% means the angle that back top is furthest from the steering wheel
	Head Support Horizontal Positon	0-100%	Get/Set/Notification	Adjust head support forward/backward, 0 means the nearest position to the front, 100% means the furthest position from the front

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Head Support Vertical Position	0-100%	Get/Set/ Notification	Adjust head support height (up or down), 0 means the lowest position, 100% means the highest position
	Seat Massaging Enabled	true, false	Get/Set/ Notification	Indicates whether message is enabled for a seat
	Message Mode	List of Struct {MessageZone, MassageMode}	Get/Set/ Notification	list of message mode of each zone
	Message Cushion Firmness	List of Struct {Cushion, 0-100%}	Get/Set/ Notification	list of firmness of each message cushion
	Seat memory	Struct{ id, label, action (SAVE/ RESTORE/ NONE)}	Get/Set/ Notification	seat memory
<b>Audio</b>	Audio volume	0%-100%	Get/Set/ Notification	The audio source volume level

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Audio Source	MOBILE_APP, RADIO_TUNE R, CD, BLUETOOTH, USB, etc. see PrimaryAudio Source	Get/Set/ Notification	defines one of the available audio sources
	keep Context	true, false	Set only	control whether HMI shall keep current application context or switch to default media UI/ APP associated with the audio source
	Equalizer Settings	Struct {Channel ID as integer, Channel setting as 0%-100%}	Get/Set/ Notification	Defines the list of supported channels (band) and their current/ desired settings on HMI
<b>Light</b>	Light Status	ON, OFF	Get/Set/ Notification	turn on/off a single light or all lights in a group

RC MODULE	CONTROL ITEM	VALUE RANGE	TYPE	COMMENTS
	Light Density	float 0.0-1.0	Get/Set/Notification	change the density/ dim a single light or all lights in a group change the color scheme of a single light or all lights in a group
	Light Color	RGB color	Get/Set/Notification	Current display mode of the HMI display
<b>HMI Settings</b>	Display Mode	DAY, NIGHT, AUTO	Get/Set/Notification	Distance Unit used in the HMI (for maps/tracking distances)
	Distance Unit	MILES, KILOMETERS	Get/Set/Notification	Temperature Unit used in the HMI (for temperature measuring systems)
	Temperature Unit	FAHRENHEIT, CELSIUS	Get/Set/Notification	

## Remote Control Button Presses

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

**RC MODULE****CONTROL BUTTON****Climate**

AC

AC MAX

RECIRCULATE

FAN UP

FAN DOWN

TEMPERATURE UP

TEMPERATURE DOWN

DEFROST

DEFROST REAR

DEFROST MAX

UPPER VENT

LOWER VENT

**Radio**

VOLUME UP

VOLUME DOWN

EJECT

SOURCE

SHUFFLE

REPEAT

# Integration

For remote control to work, the head unit must support SDL Core v.4.4 or newer. Also your app's `appType` must be set to `REMOTE_CONTROL`.

## Checking Permissions

Prior to using any remote control RPCs, you must check that the head unit has the remote control capability. As you will encounter head units that do *not* support it, this check is important. To check for this capability, use the following call:

If you do have permission to use the remote control feature, the capability object will have a list of `SDLButtonCapabilities` that can be obtained via the `buttonCapabilities` property.

### OBJECTIVE-C

```
[self.sdlManager.systemCapabilityManager updateCapabilityType:
SDLSystemCapabilityTypeRemoteControl completionHandler:^(NSError
*_Nullable error, SDLSystemCapabilityManager *_Nonnull
systemCapabilityManager) {
    if (systemCapabilityManager.remoteControlCapability == nil || error !
= nil) {
        // Remote control is not supported or capability could not be
retrieved
        return;
    }

    SDLRemoteControlCapabilities *capabilities =
systemCapabilityManager.remoteControlCapability;
    <#Code#>
}];
```

## SWIFT

```
sdIManager.systemCapabilityManager.updateCapabilityType(.
remoteControl) { (error, systemCapabilityManager) in
    if systemCapabilityManager.remoteControlCapability == nil || error !
    = nil {
        // Remote control is not supported or capability could not be
        retrieved
        return
    }

    let capabilities = systemCapabilityManager.remoteControlCapability;
    <#Code#>
}
```

## Getting Data

Once you know you have permission to use the remote control feature, you can retrieve the data. The following code is an example of how to get data from the radio module. The example also subscribes to updates to radio data, which will be discussed later on in this guide.

## OBJECTIVE-C

```
SDLGetInteriorVehicleData *getInteriorVehicleData = [[
SDLGetInteriorVehicleData alloc] initWithModuleType:
SDLModuleTypeRadio];
[self.sdIManager sendRequest:getInteriorVehicleData
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
__kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
error) {
    SDLGetInteriorVehicleDataResponse *dataResponse = (
SDLGetInteriorVehicleDataResponse *)response;
    // This can now be used to retrieve data
    <#Code#>
}];
```

## SWIFT

```
let getInteriorVehicleData = SDLGetInteriorVehicleData(
andSubscribeToModuleType: .radio)
sdlManager.send(request: getInteriorVehicleData) { (req, res, err) in
  guard let response = res as? SDLGetInteriorVehicleDataResponse
  else { return }
  // This can now be used to retrieve data
  <#Code#>
}
```

## Setting Data

Of course, the ability to set these modules is the point of the remote control framework. Setting data is similar to getting it. Below is an example of setting climate control data. It is likely that you will not need to set all the data as in the code example. If there are settings you don't wish to modify you can skip setting them.

## OBJECTIVE-C

```
SDLTemperature *temperature = [[SDLTemperature alloc] initWithUnit:
SDLTemperatureUnitFahrenheit value:74.1];
SDLClimateControlData *climateControlData = [[
SDLClimateControlData alloc] initWithFanSpeed:@2 desiredTemperature:
:temperature acEnable:@YES circulateAirEnable:@NO autoModeEnable:
@NO defrostZone:nil dualModeEnable:@NO acMaxEnable:@NO
ventilationMode:SDLVentilationModeLower heatedSteeringWheelEnable:
@YES heatedWindshieldEnable:@YES heatedRearWindowEnable:@YES
heatedMirrorsEnable:@NO];
SDLModuleData *moduleData = [[SDLModuleData alloc]
initWithClimateControlData:climateControlData];
SDLSetInteriorVehicleData *setInteriorVehicleData = [[
SDLSetInteriorVehicleData alloc] initWithModuleData:moduleData];
[self.sdlManager sendRequest:setInteriorVehicleData];
```

## SWIFT

```
let temperature = SDLTemperature(unit: .fahrenheit, value: 74.1)
let climateControlData = SDLClimateControlData(fanSpeed: 2 as
NSNumber, desiredTemperature: temperature, acEnable: true as
NSNumber, circulateAirEnable: false as NSNumber, autoModeEnable:
false as NSNumber, defrostZone: nil, dualModeEnable: false as
NSNumber, acMaxEnable: false as NSNumber, ventilationMode: .lower,
heatedSteeringWheelEnable: true as NSNumber,
heatedWindshieldEnable: true as NSNumber, heatedRearWindowEnable
: true as NSNumber, heatedMirrorsEnable: false as NSNumber)
let moduleData = SDLModuleData(climateControlData:
climateControlData)
let setInteriorVehicleData = SDLSetInteriorVehicleData(moduleData:
moduleData)

sdIManager.send(setInteriorVehicleData)
```

## Button Presses

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

## OBJECTIVE-C

```
SDLButtonPress *buttonPress = [[SDLButtonPress alloc]
initWithButtonName:SDLButtonNameEject moduleType:
SDLModuleTypeRadio];
buttonPress.buttonPressMode = SDLButtonPressModeShort;

[self.sdIManager sendRequest:buttonPress];
```

## SWIFT

```
let buttonPress = SDLButtonPress(buttonName: .eject, moduleType: .radio)
buttonPress.buttonPressMode = .short

sdIManager.send(buttonPress)
```

## Subscribing to Changes

It is also possible to subscribe to changes in data associated with supported modules. To do so, during your request for data, simply set `subscribe` to `true`. To unsubscribe, send the request again with `subscribe` set to `false`. The response to a subscription will come in a form of a notification. You can receive this notification by adding a notification listener for `OnInteriorVehicleData`.

### NOTE

The notification listener should be added before sending the `SDLGetInteriorVehicleData` request.

## OBJECTIVE-C

```
[self.sdlManager subscribeToRPC:
SDLDidReceiveInteriorVehicleDataNotification withBlock:^(__kindof
SDLRPCMessage * _Nonnull message) {
    SDLRPCNotificationNotification *dataNotification = (
SDLRPCNotificationNotification *)note;
    SDLOnInteriorVehicleData *onInteriorVehicleData = (
SDLOnInteriorVehicleData *)dataNotification.notification;
    if (onInteriorVehicleData == nil) { return; }

    // This block will now be called whenever vehicle data changes
    // NOTE: If you subscribe to multiple modules, all the data will be sent
here. You will have to split it out based on
`onInteriorVehicleData.moduleData.moduleType` yourself.
    <#Code#>
}];

SDLGetInteriorVehicleData *getInteriorVehicleData = [[
SDLGetInteriorVehicleData alloc] initWithSubscribeToModuleType:
SDLModuleTypeRadio];
[self.sdlManager sendRequest:getInteriorVehicleData
withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
__kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
error) {
    SDLGetInteriorVehicleDataResponse *dataResponse = (
SDLGetInteriorVehicleDataResponse *)response;
    // This can now be used to retrieve data
    <#Code#>
}];
```

## SWIFT

```
sdIManager.subscribe(to: .SDLDidReceiveInteriorVehicleData) { (
message) in
    let onInteriorVehicleData = message as? SDLOnInteriorVehicleData
    else { return }

    // This block will now be called whenever vehicle data changes
    // NOTE: If you subscribe to multiple modules, all the data will be
    sent here. You will have to split it out based on
    `onInteriorVehicleData.moduleData.moduleType` yourself.
    <#Code#>
}

let getInteriorVehicleData = SDLGetInteriorVehicleData(
andSubscribeToModuleType: .radio)
sdIManager.send(request: getInteriorVehicleData) { (req, res, err) in
    guard let response = res as? SDLGetInteriorVehicleDataResponse
    else { return }
    // This can now be used to retrieve initialData data
    <#Code#>
}
```

# Creating an App Service

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

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

Vehicle head units may use these services in various ways. One app service for each type will be the "active" service to the module. For media, for example, this will be the media app that the user is currently using or listening to. For

navigation, it would be a navigation app that the user is using to navigate. For weather, it may be the last used weather app, or a user-selected default. The system may then use that service's data to perform various actions (such as navigating to an address with the active service or to display the temperature as provided from the active weather service).

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

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

Using an app service is covered [in another guide](#).

## App Service Types

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

## Publishing an App Service

Publishing a service is a several step process. First, create your app service manifest. Second, publish your app service using your manifest. Third, publish your service data using `OnAppServiceData`. Fourth, respond to `GetAppServiceData` requests. Fifth, you should support RPCs related to your service. Last, optionally, you can support URI based app actions.

# 1. Creating an App Service Manifest

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

## OBJECTIVE-C

```
SDLAppServiceManifest *manifest = [[SDLAppServiceManifest alloc]
initWithAppServiceType:SDLAppServiceTypeMedia];
manifest.serviceName = @"My Media App"; // Must be unique across
app services.
manifest.servicelcon = [[SDImage alloc] initWithName:@"Service Icon
Name" isTemplate:NO]; // Previously uploaded service icon. This could
be the same as your app icon.
manifest.allowAppConsumers = @YES; // Whether or not other apps
can view your data in addition to the head unit. If set to `NO` only the
head unit will have access to this data.
manifest.rpcSpecVersion = [[SDLSyncMsgVersion alloc]
initWithMajorVersion:5 minorVersion:0 patchVersion:0]; // An *optional*
parameter that limits the RPC spec versions you can understand to the
provided version *or below*.
manifest.handledRPCs = @[]; // If you add function ids to this
*optional* parameter, you can support newer RPCs on older head units
(that don't support those RPCs natively) when those RPCs are sent
from other connected applications.
manifest.mediaServiceManifest = <#Code#> // Covered below
```

## SWIFT

```
let manifest = SDLAppServiceManifest(appServiceType: .media)
manifest.servicelcon = SDLImage(name:"Service Icon Name",
isTemplate: false) // Previously uploaded service icon. This could be the
same as your app icon.
manifest.allowAppConsumers = true; // Whether or not other apps can
view your data in addition to the head unit. If set to `NO` only the head
unit will have access to this data.
manifest.rpcSpecVersion = SDLSyncMsgVersion(majorVersion: 5,
minorVersion: 0, patchVersion: 0) // An *optional* parameter that limits
the RPC spec versions you can understand to the provided version *or
below*.
manifest.handledRPCs = []; // If you add function ids to this *optional*
parameter, you can support newer RPCs on older head units (that don't
support those RPCs natively) when those RPCs are sent from other
connected applications.
manifest.mediaServiceManifest = <#Code#> // Covered below
```

---

## CREATING A MEDIA SERVICE MANIFEST

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

## OBJECTIVE-C

```
SDLMediaServiceManifest *mediaManifest = [[
SDLMediaServiceManifest alloc] init];
manifest.mediaServiceManifest = mediaManifest;
```

## SWIFT

```
let mediaManifest = SDLMediaServiceManifest()  
manifest.mediaServiceManifest = mediaManifest
```

---

## CREATING A NAVIGATION SERVICE MANIFEST

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

## OBJECTIVE-C

```
SDLNavigationServiceManifest *navigationManifest = [[  
SDLNavigationServiceManifest alloc] initWithAcceptsWayPoints:YES];  
manifest.navigationServiceManifest = navigationManifest;
```

## SWIFT

```
let navigationManifest = SDLNavigationServiceManifest(  
acceptsWayPoints: true)  
manifest.navigationServiceManifest = navigationManifest
```

---

## CREATING A WEATHER SERVICE MANIFEST

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

### OBJECTIVE-C

```
SDLWeatherServiceManifest *weatherManifest = [[
SDLWeatherServiceManifest alloc] initWithCurrentForecastSupported:
YES maxMultidayForecastAmount:10 maxHourlyForecastAmount:24
maxMinutelyForecastAmount:60 weatherForLocationSupported:YES];
manifest.weatherServiceManifest = weatherManifest;
```

### SWIFT

```
let weatherManifest = SDLWeatherServiceManifest(
currentForecastSupported: true, maxMultidayForecastAmount: 10,
maxHourlyForecastAmount: 24, maxMinutelyForecastAmount: 60,
weatherForLocationSupported: true)
manifest.weatherServiceManifest = weatherManifest
```

## 2. Publish Your Service

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

## OBJECTIVE-C

```
SDLPublishAppService *publishServiceRequest = [[
    SDLPublishAppService alloc] initWithAppServiceManifest:<#Manifest
    Object#>];
[self.sdlManager sendRequest:publishServiceRequest
    withResponseHandler:^(__kindof SDLRPCRequest * _Nullable request,
    __kindof SDLRPCResponse * _Nullable response, NSError * _Nullable
    error) {
    if (error != nil || !response.success.boolValue) { return; }

    SDLPublishAppServiceResponse *publishServiceResponse = (
    SDLPublishAppServiceResponse *)response;
    SDLAppServiceRecord *serviceRecord = publishServiceResponse.
    appServiceRecord
    <#Use the response#>
}];
```

## SWIFT

```
let publishServiceRequest = SDLPublishAppService(appServiceManifest:
    <#Manifest Object#>)
sdlManager.send(request: publishServiceRequest) { (req, res, err) in
    guard let response = res as? SDLPublishAppServiceResponse,
    response.success.boolValue == true, err == nil else { return }

    let serviceRecord = response.appServiceRecord
    <#Use the response#>
}
```

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

---

## WATCHING FOR APP RECORD UPDATES

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

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

For more information, see the [Using App Services guide](#) and see the "Getting and Subscribing to Services" section.

### 3. Update Your Service's Data

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

#### NOTE

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

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

---

## MEDIA SERVICE DATA

### OBJECTIVE-C

```
SDLMediaServiceData *mediaData = [[SDLMediaServiceData alloc]
initWithMediaType:SDLMediaTypeMusic mediaTitle:@"Some media title"
mediaArtist:@"Some media artist" mediaAlbum:@"Some album"
playlistName:@"Some playlist" isExplicit:YES trackPlaybackProgress:45
trackPlaybackDuration:90 queuePlaybackProgress:45
queuePlaybackDuration:150 queueCurrentTrackNumber:2
queueTotalTrackCount:3];
SDLAppServiceData *appData = [[SDLAppServiceData alloc]
initWithMediaServiceData:mediaData serviceld:myServiceld];

SDLOnAppServiceData *onAppData = [[SDLOnAppServiceData alloc]
initWithServiceData:appData];
[self.sdlManager sendRPC:onAppData];
```

### SWIFT

```
let mediaData = SDLMediaServiceData(mediaType: .music, mediaTitle:
"Some media title", mediaArtist: "Some artist", mediaAlbum: "Some
album", playlistName: "Some playlist", isExplicit: true,
trackPlaybackProgress: 45, trackPlaybackDuration: 90,
queuePlaybackProgress: 45, queuePlaybackDuration: 150,
queueCurrentTrackNumber: 2, queueTotalTrackCount: 3)
let appMediaData = SDLAppServiceData(mediaServiceData: mediaData
, serviceld: serviceld)

let onAppData = SDLOnAppServiceData(serviceData: appMediaData)
sdlManager.sendRPC(onAppData)
```

---

## NAVIGATION SERVICE DATA

### OBJECTIVE-C

```
UIImage *image = [UIImage imageNamed:imageName];
if (image == nil) { return; }

SDLArtwork *artwork = [SDLArtwork artworkWithImage:image name:
imageName asImageFormat:SDLArtworkImageFormatJPG];
[self.sdlManager.fileManager uploadFile:artwork completionHandler:^(
BOOL success, NSUInteger bytesAvailable, NSError * _Nullable error) {
    // Make sure the image is uploaded to the system before publishing
    your service
    SDLLocationCoordinate *coordinate = [[SDLLocationCoordinate alloc]
initWithLatitudeDegrees:42 longitudeDegrees:43];
    SDLLocationDetails *location = [[SDLLocationDetails alloc]
initWithCoordinate:coordinate];
    SDLNavigationInstruction *instruction = [[SDLNavigationInstruction
alloc] initWithLocationDetails:location action:SDLNavigationActionTurn];
    instruction.image = [[SDLImage alloc] initWithName:imageName
isTemplate:NO];

    SDLDateTime *timestamp = [[SDLDateTime alloc] initWithHour:2
minute:3 second:4 millisecond:847];
    SDLNavigationServiceData *navServiceData = [[
SDLNavigationServiceData alloc] initWithTimestamp:timestamp];
    navServiceData.instructions = @[instruction];

    SDLAppServiceData *appServiceData = [[SDLAppServiceData alloc]
initWithNavigationServiceData:navServiceData serviceId:@"<#Your
saved serviceID#>"];

    SDLOnAppServiceData *onAppServiceData = [[
SDLOnAppServiceData alloc] initWithServiceData:appServiceData];
    [self.sdlManager sendRPC:onAppServiceData];
}];
```

## SWIFT

```
guard let image = UIImage(named: imageName) else { return }
let artwork = SDLArtwork(image: image, name: imageName, persistent
: false, as: .JPG)

sdIManager.fileManager.upload(file: artwork) { [weak self] (success,
bytesAvailable, error) in
    guard success else { return }

    // Make sure the image is uploaded to the system before publishing
your service
    let coordinate = SDLLocationCoordinate(latitudeDegrees: 42,
longituddeDegrees: 43)
    let location = SDLLocationDetails(coordinate: coordinate)
    let instruction = SDLNavigationInstruction(locationDetails: location,
action: .turn)
    instruction.image = SDLImage(name: imageName, isTemplate: false)

    let timestamp = SDLDateTime(hour: 2, minute: 3, second: 4,
millisecond: 847)
    let navServiceData = SDLNavigationServiceData(timestamp:
timestamp)
    navServiceData.instructions = [instruction]

    let appServiceData = SDLAppServiceData(navigationServiceData:
navServiceData, serviceId: "<#Your saved serviceID#>")

    let onAppServiceData = SDLOnAppServiceData(serviceData:
appServiceData)
    self?.sdIManager.sendRPC(onAppServiceData)
}
```

---

## WEATHER SERVICE DATA

## OBJECTIVE-C

```
UIImage *image = [UIImage imageNamed:imageName];
if (image == nil) { return; }

SDLArtwork *artwork = [SDLArtwork artworkWithImage:image name:
imageName asImageFormat:SDLArtworkImageFormatJPG];

// We have to send the image to the system before it's used in the app
service.
__weak typeof(self) weakSelf = self;
[self.sdlManager.fileManager uploadFile:artwork completionHandler:^(
BOOL success, NSUInteger bytesAvailable, NSError * _Nullable error) {
    [weakSelf updateWeatherServiceWithImage:success];
}];

- (void)updateWeatherServiceWithImage:(BOOL)useImage {
    SDLWeatherData *weatherData = [[SDLWeatherData alloc] init];
    weatherData.weatherIconImageName = useImage ? imageName : nil
;

    SDLWeatherServiceData *weatherServiceData = [[
SDLWeatherServiceData alloc] initWithLocation:[[SDLLocationDetails
alloc] initWithCoordinate:[[SDLLocationCoordinate alloc]
initWithLatitudeDegrees:42.331427 longitudeDegrees:-83.0457538]];

    SDLAppServiceData *appServiceData = [[SDLAppServiceData alloc]
initWithWeatherServiceData:weatherServiceData serviceId:@"<#Your
saved serviceID#>"];

    SDLOnAppServiceData *onAppServiceData = [[
SDLOnAppServiceData alloc] initWithServiceData:appServiceData];
    [self.sdlManager sendRPC:onAppServiceData];
}
```

## SWIFT

```
guard let image = UIImage(named: imageName) else { return }
let artwork = SDLArtwork(image: image, name: imageName, persistent
: false, as: .JPG)

// We have to send the image to the system before it's used in the app
service.
sdManager.fileManager.upload(file: artwork) { [weak self] (success,
bytesAvailable, error) in
    self?.updateWeatherService(shouldUseImage: success)
}

private func updateWeatherService(shouldUseImage: Bool) {
    let weatherData = SDLWeatherData()
    weatherData.weatherIconImageName = shouldUseImage ?
imageName : nil

    let weatherServiceData = SDLWeatherServiceData(location:
SDLLocationDetails(coordinate: SDLLocationCoordinate(latitudeDegrees
: 42.3314, longitudeDegrees: 83.0458)), currentForecast: weatherData,
minuteForecast: nil, hourlyForecast: nil, multidayForecast: nil, alerts: nil
)

    let appServiceData = SDLAppServiceData(weatherServiceData:
weatherServiceData, serviceId: "<#Your saved serviceID#>")

    let onAppServiceData = SDLOnAppServiceData(serviceData:
appServiceData)
    sdManager.sendRPC(onAppServiceData)
}
```

## 4. Handling App Service Subscribers

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

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

---

## LISTENING FOR REQUESTS

First, you will need to register for notification of a `GetAppServiceDataRequest`.

### OBJECTIVE-C

```
// sdl_ios v6.3+
[self.sdlManager subscribeToRPC:
SDLDidReceiveGetAppServiceDataRequest withBlock:^(__kindof
SDLRPCMessage * _Nonnull message) {
    SDLGetAppServiceData *getAppServiceRequest = message;

    <#Use the request#>
}];

// Pre sdl_ios v6.3
[[NSNotificationCenter defaultCenter] addObserver:self selector:
@selector(appServiceDataRequestReceived:) name:
SDLDidReceiveGetAppServiceDataRequest object:nil];
```

### SWIFT

```
// sdl_ios v6.3+
sdlManager.subscribe(to: SDLDidReceiveGetAppServiceDataRequest) {
(message) in
    guard let getAppServiceRequest = message as?
SDLGetAppServiceData else { return }

    <#Use the request#>
}

// Pre sdl_ios v6.3
NotificationCenter.default.addObserver(self, selector: #selector(
appServiceDataRequestReceived(_:)), name:
SDLDidReceiveGetAppServiceDataRequest, object: nil)
```

---

## SENDING A RESPONSE TO SUBSCRIBERS

Second, you need to respond to the request when you receive it with your app service data. This means that you will need to store your current service data after your most recent update using `OnAppServiceData` (see the section [Updating Your Service Data](#)).

### OBJECTIVE-C

```
- (void)appServiceDataRequestReceived:(SDLRPCRequestNotification *)
request {
    SDLGetAppServiceData *getAppServiceData = (
SDLGetAppServiceData *)request.request;

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

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

## SWIFT

```
@objc func appServiceDataRequestReceived(_ request:
SDLRPCRequestNotification) {
    guard let getAppServiceData = request.request as?
SDLGetAppServiceData else { return }

    // Send a response
    let response = SDLGetAppServiceDataResponse(appServiceData: <#
Your App Service Data#>)
    response.correlationID = getAppServiceData.correlationID
    response.success = true as NSNumber
    response.resultCode = .success
    response.info = "<#Use to provide more information about an
error#>"

    sdlManager.sendRPC(response)
}
```

# Supporting Service RPCs and Actions

## 5. Service RPCs

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

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

When you are the active service for your service's type (e.g. media), and you have declared that you support these RPCs in your manifest (see section 1. Creating an App Service Manifest), then these RPCs will be automatically routed to your app. You will have to set up notifications to be aware that they have arrived, and you will then need to respond to those requests.

## OBJECTIVE-C

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

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

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

    // Send a response
    SDLButtonPressResponse *response = [[SDLButtonPressResponse
alloc] init];
    response.correlationID = buttonPressRequest.correlationID;
    response.success = @YES;
    response.resultCode = SDLVehicleDataResultCodeSuccess;
    response.info = @"<#Use to provide more information about an
error#>";

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

## SWIFT

```
let manifest = SDLAppServiceManifest()
// Everything else for your manifest
let buttonPressRPCID = SDLFunctionID.sharedInstance().functionId(
  forName: .buttonPress)
manifest.handledRPCs = [buttonPressRPCID]

sdIManager.subscribe(to: SDLDidReceiveButtonPressRequest, observer:
  self, selector: #selector(buttonPressRequestReceived(_:)))

@objc private func buttonPressRequestReceived(_ notification:
  SDLRPCRequestNotification) {
  guard let interactionRequest = notification.request as?
  SDLButtonPress else { return }

  // A result you want to send to the consumer app.
  let response = SDLButtonPressResponse()

  // These are very important, your response won't work properly
  without them.
  response.success = true
  response.resultCode = .success
  response.correlationID = interactionRequest.correlationID
  response.info = "<#Use to provide more information about an
  error#>"

  sdIManager.sendRPC(response)
}
```

## 6. Service Actions

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

If you're wondering how to get started with actions and routing, this is a very common problem in iOS! Many apps support the [x-callback-URL](#) format as a

common inter-app communication method. There are also [many libraries available](#) for the purpose of supporting URL routing.

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

## OBJECTIVE-C

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

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

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

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

    // The URL sent by the consumer. This must be something you
understand, e.g. a URL scheme call. For example, if you were Youtube,
it could be a URL to play a specific video. If you were a music app, it
could be a URL to play a specific song, activate shuffle / repeat, etc.
    NSURLComponents *interactionURLComponents = [
NSURLComponents componentsWithString:interactionRequest.
serviceUri];

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

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

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

## SWIFT

```
// Subscribe to PerformAppServiceInteraction requests
sdIManager.subscribe(to:
SDLDidReceivePerformAppServiceInteractionRequest, observer: self,
selector: #selector(performAppServiceInteractionRequestReceived(_:)))

@objc private func performAppServiceInteractionRequestReceived(_
notification: SDLRPCRequestNotification) {
    guard let interactionRequest = notification.request as?
SDLPerformAppServiceInteraction else { return }

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

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

    // The URL sent by the consumer. This must be something you
understand, e.g. a URL scheme call. For example, if you were Youtube,
it could be a URL to play a specific video. If you were a music app, it
could be a URL to play a specific song, activate shuffle / repeat, etc.
    let interactionURLComponents = URLComponents(string:
interactionRequest.serviceUri)

    // A result you want to send to the consumer app.
    let result = "Uh oh"
    let response = SDLPerformAppServiceInteractionResponse(
serviceSpecificResult: result)

    // These are very important, your response won't work properly
without them.
    response.success = true
    response.resultCode = .success
    response.correlationID = interactionRequest.correlationID

    sdIManager.sendRPC(response)
}
```

# Using App Services

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

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

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

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

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

# Getting and Subscribing to Services

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

## 1. Getting and Subscribing to Available Services

To get information on all services published on the system, as well as on changes to published services, you will use the `GetSystemCapability` request / response as well as the `OnSystemCapabilityUpdated` notification.

## OBJECTIVE-C

```
// sdl_ios v6.3+
id subscribedObserver = [self.sdlManager.systemCapabilityManager
subscribeToCapabilityType:SDLSystemCapabilityTypeAppServices
withBlock:^(SDLSystemCapability * _Nonnull capability) {
    NSArray<SDLAppServicesCapabilities *> *appServices = capability.
appServicesCapabilities.appServices;

    <#Use the app services records#>
}];

// Pre sdl_ios v6.3
- (void)systemCapabilityDidUpdate:(SDLRPCNotificationNotification *)
notification {
    SDLOnSystemCapabilityUpdated *updateNotification = notification.
notification;
    SDLLogD(@"On System Capability updated: %@", updateNotification
);

    <#Use the updated services records#>
}

- (void)setupAppServicesCapability {
    [self.sdlManager subscribeToRPC:
SDLDidReceiveSystemCapabilityUpdatedNotification withObserver:self
selector:@selector(systemCapabilityDidUpdate:)];

    SDLGetSystemCapability *getAppServices = [[
SDLGetSystemCapability alloc] initWithType:
SDLSystemCapabilityTypeAppServices subscribe:YES];
    [self.sdlManager sendRequest:getAppServices withResponseHandler:
^(__kindof SDLRPCRequest * _Nullable request, __kindof
SDLRPCResponse * _Nullable response, NSError * _Nullable error) {
        if (!response || !response.success.boolValue) {
            SDLLogE(@"Error sending get system capability: Req %@, Res
%@, err %@", request, response, error);
            return;
        }
    }

    SDLAppServicesCapabilities *serviceRecord = response.
systemCapability.appServicesCapabilities
    SDLLogD(@"Get system capability app service response: %@,
serviceRecord %@", response, serviceRecord);

    <#Use the service record#>
}];
}
```

## SWIFT

```
// sdl_ios v6.3+
let subscribedObserver = sdlManager.systemCapabilityManager.
subscribe(toCapabilityType: .appServices) { (systemCapability) in
    let appServices = systemCapability.appServicesCapabilities?.
appServices

    <#Use the app services records#>
}

// Pre sdl_ios v6.3
@objc private func systemCapabilityDidUpdate(_ notification:
SDLRPCNotificationNotification) {
    guard let capabilityNotification = notification.notification as?
SDLOnSystemCapabilityUpdated else { return }

    SDLLog.d("OnSystemCapabilityUpdated: \(capabilityNotification)")
    <#Use the updated services records#>
}

private func setupAppServicesCapability() {
    Notification.default.addObserver(self, selector: #selector(
systemCapabilityDidUpdate(_:)), name: .
SDLDidReceiveSystemCapabilityUpdated, object: nil)

    let getAppServices = SDLGetSystemCapability(type: .appServices,
subscribe: true)
    sdlManager.send(request: getAppServices) { (req, res, err) in
        guard let response = res as? SDLGetSystemCapabilityResponse,
let serviceRecord = response.systemCapability.appServicesCapabilities,
response.success.boolValue == true, err == nil else { return }

        <#Use the service record#>
    }
}
```

---

## CHECKING THE APP SERVICE CAPABILITY

Once you've retrieved the initial list of app service capabilities (in the `GetSystemCapability` response), or an updated list of app service capabilities (from the `OnSystemCapabilityUpdated` notification), you may want to inspect the data

to find what you are looking for. Below is example code with comments explaining what each part of the app service capability is used for.

## OBJECTIVE-C

```
// From GetSystemCapabilityResponse
SDLGetSystemCapabilityResponse *getResponse = <#From wherever
you got it#>;
SDLAppServicesCapabilities *capabilities = getResponse.
systemCapability.appServicesCapabilities;

// This array contains all currently available app services on the system
NSArray<SDLAppServiceCapability *> *appServices = capabilities.
appServices;
SDLAppServiceCapability *aCapability = appServices.first;

// This will be nil since it's the first update
SDLServiceUpdateReason *capabilityReason = aCapability.
updateReason;

// The app service record will give you access to a service's generated
id, which can be used to address the service directly (see below), it's
manifest, used to see what data it supports, whether or not the service
is published (it always will be here), and whether or not the service is
the active service for its service type (only one service can be active
for each type)
SDLAppServiceRecord *serviceRecord = aCapability.
updatedAppServiceRecord;

// From OnSystemCapabilityUpdated
SDLOnSystemCapabilityUpdated *serviceNotification = <#From
wherever you got it#>;
SDLAppServicesCapabilities *capabilities = serviceNotification.
systemCapability.appServicesCapabilities;

// This array contains all recently updated services
NSArray<SDLAppServiceCapability *> *appServices = capabilities.
appServices;
SDLAppServiceCapability *aCapability = appServices.first;

// This won't be nil. It will tell you why a service is in the list of updates
SDLServiceUpdateReason *capabilityReason = aCapability.
updateReason;

// The app service record will give you access to a service's generated
id, which can be used to address the service directly (see below), it's
manifest, used to see what data it supports, whether or not the service
is published (if it's not, it was just removed and should not be
addressed), and whether or not the service is the active service for its
service type (only one service can be active for each type)
SDLAppServiceRecord *serviceRecord = aCapability.
updatedAppServiceRecord;
```

## SWIFT

```
// From GetSystemCapabilityResponse
let getResponse: SDLGetSystemCapabilityResponse = <#From
wherever you got it#>;
let capabilities = getResponse.systemCapability.
appServicesCapabilities;

// This array contains all currently available app services on the system
let appServices: [SDLAppServiceCapability] = capabilities.appServices
let aCapability = appServices.first;

// This will be nil since it's the first update
let capabilityReason = aCapability.updateReason;

// The app service record will give you access to a service's generated
id, which can be used to address the service directly (see below), it's
manifest, used to see what data it supports, whether or not the service
is published (it always will be here), and whether or not the service is
the active service for its service type (only one service can be active
for each type)
let serviceRecord = aCapability.updatedAppServiceRecord;

// From OnSystemCapabilityUpdated
let serviceNotification: SDLOnSystemCapabilityUpdated = <#From
wherever you got it#>;
let capabilities = serviceNotification.systemCapability.
appServicesCapabilities;

// This array contains all recently updated services
let appServices: [SDLAppServiceCapability] = capabilities.appServices;
let aCapability = appServices.first;

// This won't be nil. It will tell you why a service is in the list of updates
let capabilityReason = aCapability.updateReason;

// The app service record will give you access to a service's generated
id, which can be used to address the service directly (see below), it's
manifest, used to see what data it supports, whether or not the service
is published (if it's not, it was just removed and should not be
addressed), and whether or not the service is the active service for its
service type (only one service can be active for each type)
let serviceRecord = aCapability.updatedAppServiceRecord;
```

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

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

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

## OBJECTIVE-C

```
// Get service data once
SDLGetAppServiceData *getServiceData = [[SDLGetAppServiceData
alloc] initWithServiceType:SDLAppServiceTypeMedia];

// Subscribe to service data in perpetuity via `OnAppServiceData`
notifications.
SDLGetAppServiceData *subscribeServiceData = [[
SDLGetAppServiceData alloc] initAndSubscribeToServiceType:
SDLAppServiceTypeMedia];

// Unsubscribe to service data previously subscribed
SDLGetAppServiceData *unsubscribeServiceData = [[
SDLGetAppServiceData alloc] initWithServiceType:
SDLAppServiceTypeMedia];
unsubscribeServiceData.subscribe = NO;

// Get the service's data
[self.sdlManager sendRequest:getServiceData withResponseHandler:^(
__typeof SDLRPCRequest * _Nullable request, __typeof SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error sending get system capability: Req %@, Res %
@, err %@", request, response, error);
        return;
    }

    SDLGetAppServiceDataResponse *serviceResponse = (
SDLGetAppServiceDataResponse *)response;
    SDLMediaServiceData *mediaData = serviceResponse.serviceData.
mediaServiceData;
}];
```

## SWIFT

```
// Get service data once
let getServiceData = SDLGetAppServiceData(serviceType: .media)

// Subscribe to service data in perpetuity via `OnAppServiceData`
notifications.
let subscribeServiceData = SDLGetAppServiceData(
andSubscribeToAppServiceType: .media)

// Unsubscribe to service data previously subscribed
let unsubscribeServiceData = SDLGetAppServiceData(serviceType: .
media)
unsubscribeServiceData.subscribe = false

// Get the service's data
let getServiceData = SDLGetAppServiceData(serviceType: .media)
sdlManager.send(request: getServiceData) { (req, res, err) in
    guard let response = res as? SDLGetAppServiceDataResponse,
response.success.boolValue == true, err == nil, let mediaData =
response.serviceData.mediaData else { return }

    <#Use the mediaData#>
}
```

# Interacting with a Service Provider

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

## 3. Sending RPCs to a Service Provider

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

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

## NOTE

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

## OBJECTIVE-C

```
SDLButtonPress *buttonPress = [[SDLButtonPress alloc]
initWithButtonName:SDLButtonNameOk moduleType:
SDLModuleTypeAudio];

[self.sdlManager sendRequest:buttonPress withResponseHandler:^(
__typeof SDLRPCRequest * _Nullable request, __typeof SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error sending button press: Req %@, Res %@, err %
@", request, response, error);
        return;
    }

    SDLButtonPressResponse *pressResponse = (
SDLButtonPressResponse *)response;
    <#Use the response#>
}];
```

## SWIFT

```
let buttonPress = SDLButtonPress(buttonName: .ok, moduleType: .audio
)
sdIManager.send(request: getServiceData) { (req, res, err) in
    guard let response = res as? SDLButtonPressResponse, response.
success.boolValue == true, err == nil else { return }

    <#Use the response#>
}
```

## 4. Sending an Action to a Service Provider

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

## OBJECTIVE-C

```
SDLPerformAppServiceInteraction *performAction = [[
SDLPerformAppServiceInteraction alloc] initWithServiceURI:
@"sdlexample://x-callback-url/showText?x-source=MyApp&text=My%
20Custom%20String" serviceID: <#Previously Retrived ServiceID#>
originApp: <#Your App Id#> requestServiceActive: NO];
[self.sdIManager sendRequest:performAction withResponseHandler:^(
__typeof SDLRPCRequest * _Nullable request, __typeof SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error sending perform action: Req %@, Res %@, err %
@" , request, response, error);
        return;
    }

    SDLPerformAppServiceInteractionResponse *actionResponse = (
SDLPerformAppServiceInteractionResponse *)response;
    <#Use the response#>
}];
```

## SWIFT

```
let performAction = SDLPerformAppServiceInteraction(serviceUri:
"sdlexample://x-callback-url/showText?x-source=MyApp&text=My%
20Custom%20String", serviceID: <#Previously Retrived ServiceID#>,
originApp: <#Your App Id#>, requestServiceActive: false)
sdIManager.send(request: performAction) { (req, res, err) in
    guard let response = res as?
SDLPerformAppServiceInteractionResponse else { return }

    <#Check the error and response#>
}
```

# Calling a Phone Number

The `SDLDialNumber` RPC allows you make a phone call via the user's phone. Regardless of platform (Android or iOS), you must be sure that a device is connected via Bluetooth (even if using USB) for this RPC to work. If the phone is not connected via Bluetooth, you will receive a result of `REJECTED` from Core.

### NOTE

`SDLDialNumber` 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.

# Checking if Dial Number is Available

`SDLDialNumber` is a newer RPC, so there is a possibility that not all head units will support it. To find out if the RPC is supported by the head unit, check the system capability manager's `hmiCapabilities.phoneCall` property after the manager has been started successfully.

## OBJECTIVE-C

```
[self.sdlManager.systemCapabilityManager updateCapabilityType:
SDLSystemCapabilityTypePhoneCall completionHandler:^(NSError *
_Nullable error, SDLSystemCapabilityManager * _Nonnull
systemCapabilityManager) {
    BOOL isDialNumberSupported = NO;
    if (error == nil) {
        isDialNumberSupported = systemCapabilityManager.
phoneCapability.dialNumberEnabled.boolValue;
    }
    else {
        isDialNumberSupported = systemCapabilityManager.
hmiCapabilities.phoneCall.boolValue;
    }

    <#If making phone calls is supported, send the `DialNumber` RPC#
>
}];
```

## SWIFT

```
sdIManager.systemCapabilityManager.updateCapabilityType(.phoneCall
) { (error, systemCapabilityManager) in
    var isDialNumberSupported = false
    if error == nil {
        isDialNumberSupported = systemCapabilityManager.
phoneCapability?.dialNumberEnabled?.boolValue ?? false;
    } else {
        isDialNumberSupported = systemCapabilityManager.
hmiCapabilities?.phoneCall?.boolValue ?? false
    }

    <#If making phone calls is supported, send the `DialNumber` RPC#
>
}
```

## Sending a DialNumber Request

### NOTE

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

## OBJECTIVE-C

```
SDLPhoneNumber *phoneNumber = [[SDLPhoneNumber alloc] initWithNumber
: @"1238675309"];

[self.sdManager sendRequest:dialNumber withResponseHandler:^(
__kindof SDLRPCRequest * _Nullable request, __kindof SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (error != nil || ![response isKindOfClass:SDLPhoneNumberResponse.
class]) {
        <#Encountered error sending DialNumber#>
        return;
    }

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

    <#DialNumber successfully sent#>
}];
```

## SWIFT

```
let dialNumber = SDL DialNumber(number: "1238675309")

sdManager.send(request: dialNumber) { (request, response, error) in
    guard let response = response as? SDL DialNumberResponse, error
    == nil else {
        <#Encountered error sending DialNumber#>
        return
    }

    guard response.resultCode == .success else {
        switch response.resultCode {
            case .rejected:
                <#DialNumber was rejected. Either the call was sent and
                cancelled or there is no device connected#>
            case .disallowed:
                <#Your app is not allowed to use DialNumber#>
            default:
                <#Some unknown error has occurred#>
        }
        return
    }

    <#DialNumber 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

The `SendLocation` RPC gives you the ability to send a GPS location to the active navigation app on the head unit.

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

## Checking If Your App Has Permission to Use Send Location

The `SendLocation` RPC is restricted by most vehicle manufacturers. As a result, the head unit you are connecting to will reject the request if you do not have the correct permissions. Please check the [Understanding Permissions](#) section for more information on how to check permissions for an RPC.

## Checking if Head Unit Supports Send Location

Since there is a possibility that some head units will not support the send location feature, you should check head unit support before attempting to send the request. You should also update your app's UI based on whether or not you can use `SendLocation`.

If using library v.6.0 and connecting to SDL Core v.4.5 or newer, you can use the `SDLSystemCapabilityManager` to check the navigation capability returned by Core as shown in the code sample below.

If connecting to older versions of Core (or using older versions of the library), you will have to check the `SDLManager.registerResponse.hmiCapabilities.navigation` after the SDL app has started successfully to see if an embedded navigation system is available. If it is, then you can assume that `SendLocation` will work.

## OBJECTIVE-C

```
[self.sdlManager.systemCapabilityManager updateCapabilityType:
SDLSystemCapabilityTypeNavigation completionHandler:^(NSError *
_Nullable error, SDLSystemCapabilityManager * _Nonnull
systemCapabilityManager) {
    BOOL isNavigationSupported = NO;
    if (error == nil) {
        isNavigationSupported = systemCapabilityManager.
navigationCapability.sendLocationEnabled.boolValue;
    }
    else {
        isNavigationSupported = systemCapabilityManager.
hmiCapabilities.navigation.boolValue;
    }

    <#If navigation is supported, send the `SendLocation` RPC#>
}];
```

## SWIFT

```
sdIManager.systemCapabilityManager.updateCapabilityType(.
navigation) { (error, systemCapabilityManager) in
    var isNavigationSupported = false
    if error == nil {
        isNavigationSupported = systemCapabilityManager.
navigationCapability?.sendLocationEnabled?.boolValue ?? false;
    } else {
        isNavigationSupported = systemCapabilityManager.
hmiCapabilities?.navigation?.boolValue ?? false
    }

    <#If navigation is supported, send the `SendLocation` RPC#>
}
```

## Using Send Location

To use the `SendLocation` request, you must at minimum 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:^(
__typeof(SDLRPCRequest * _Nullable request, __typeof(SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (error || ![response isKindOfClass:SDLSendLocationResponse.class])
    {
        <#Encountered error sending SendLocation#>
        return;
    }

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

    <#SendLocation successfully sent#>
}]);
```

## 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)

sdIManager.send(request: sendLocation) { (request, response, error) in
    guard let response = response as? SDLSendLocationResponse, error
    == nil else {
        <#Encountered error sending SendLocation#>
        return
    }

    guard response.resultCode == .success else {
        switch response.resultCode {
        case .invalidData:
            <#SendLocation was rejected. The request contained invalid
data.#>
        case .disallowed:
            <#Your app is not allowed to use SendLocation#>
        default:
            <#Some unknown error has occurred#>
        }
        return
    }

    <#SendLocation successfully sent#>
}
```

# Checking the Result of Send Location

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

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

# Uploading Files

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

## Uploading an MP3 Using the File Manager

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

### OBJECTIVE-C

```
NSData *mp3Data = <#Get the File Data#>;
SDLFile *audioFile = [SDLFile fileWithData:mp3Data name:<#File name
#> fileExtension:<#File Extension#>];

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

## SWIFT

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

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

# Batching File Uploads

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

## 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#
> fileExtension:<#File Extension#>];

[self.sdIManager.fileManager uploadFiles:@[file1, file2] progressHandler
:^BOOL(NSString * _Nonnull fileName, float uploadPercentage, NSError
* _Nullable error) {
    <#Called as each upload completes#>
    // Return true to continue sending files. Return false to cancel any
files that have not yet been sent.
    return YES;
} completionHandler:^(NSArray<NSString *> * _Nonnull fileNames,
NSError * _Nullable error) {
    <#Called when all uploads complete#>
}];
```

## 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#>)

sdIManager.fileManager.upload(files: [file1, file2], progressHandler: { (
fileName, uploadPercentage, error) -> Bool in
  <#Called as each upload completes#>
  // Return true to continue sending files. Return false to cancel any
files that have not yet been sent.
  return true
}) { (fileNames, error) in
  <#Called when all uploads complete#>
}
```

## File Persistence

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

### OBJECTIVE-C

```
BOOL isPersistent = file.isPersistent;
```

## SWIFT

```
let isPersistent = file.isPersistent;
```

### NOTE

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

# Overwriting Stored Files

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

## OBJECTIVE-C

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

## SWIFT

```
file.overwrite = true
```

# Checking the Amount of File Storage Left

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

## OBJECTIVE-C

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

## SWIFT

```
let bytesAvailable = sdManager.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.sdFileManager.fileManager.  
remoteFileNames containsObject:<#Name Uploaded As#>];
```

## SWIFT

```
let isFileOnHeadUnit = sdFileManager.fileManager.remoteFileNames.  
contains(<#Name Uploaded As#>)
```

# Deleting Stored Files

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

## OBJECTIVE-C

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

## SWIFT

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

# Batch Deleting Files

## OBJECTIVE-C

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

## SWIFT

```
sdFileManager.fileManager.delete(fileNames: ["<#Name Uploaded As#>",  
    "<#Name Uploaded As 2#>"]) { (error) in  
    if (error == nil) {  
        <#Files 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. Persistent images are stored on a head unit between sessions. Ephemeral images are destroyed when a session ends (i.e. when the user turns off their vehicle).
4. Images can not be uploaded when the app's `hmiLevel` is `NONE`. For more information about permissions, please review [Understanding Permissions](#).

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 unnecessary image data. To check if graphics are supported, look at the `SDLManager`'s `systemCapabilityManager`'s `displayCapabilities` property once the `SDLManager` has started successfully.

## OBJECTIVE-C

```
__weak typeof(self) weakSelf = self;
[self.sdlManager startWithReadyHandler:^(BOOL success, NSError *
_nullable error) {
    if (!success) {
        <#Manager errored while starting up#>
        return;
    }

    SDLDisplayCapabilities *displayCapabilities = weakSelf.sdlManager.
systemCapabilityManager.displayCapabilities;
    BOOL graphicsSupported = (displayCapabilities != nil) ?
displayCapabilities.graphicSupported.boolValue : NO;
}];
```

## SWIFT

```
sdFileManager.start { [weak self] (success, error) in
    guard let self = self else { return }

    guard success else {
        <#Manager errored while starting up#>
        return
    }

    let displayCapabilities = self.sdFileManager.systemCapabilityManager.
displayCapabilities
    let graphicsSupported = displayCapabilities != nil ?
displayCapabilities?.graphicSupported.boolValue : false
}
```

# Uploading an Image Using SDL FileManager

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 persistentArtworkWithImage:
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
isTemplate:<#BOOL#>];
}];
```

## 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, isTemplate: <#Bool#>
)
}
```

## Batch File Uploads, Persistence, etc.

Similar to other files, artworks can be persistent, batched, overwrite, etc. See [Uploading Files](#) for more information.

# Creating an OEM Cloud App Store

A new feature of SDL Core v5.1 and Sdl Java Suite v.6.2 allows OEMs to offer an app store that lets users browse and install remote cloud apps. If the cloud app requires users to login with their credentials, the app store can use an authentication token to automatically login users after their first session.

### NOTE

An OEM app store can be a mobile app or a cloud app.

## User Authentication

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

# Setting and Getting Cloud App Properties

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

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

## NOTE

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

## Setting Cloud App Properties

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

### OBJECTIVE-C

```
SDLCloudAppProperties *properties = [[SDLCloudAppProperties alloc]
initWithAppID:<#app id#>];
properties.authToken = <#auth token#>;
SDLSetCloudAppProperties *setCloud = [[SDLSetCloudAppProperties
alloc] initWithProperties:properties];
[self.sdlManager sendRequest:setCloud withResponseHandler:^(
__typeof(SDLRPCRequest * _Nullable request, __typeof(SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error sending set cloud properties: Req %@, Res %@,
err %@", request, response, error);
        return;
    }

    SDLSetCloudAppPropertiesResponse *setCloudResponse = (
SDLSetCloudAppPropertiesResponse *)response;

    <#Use the response#>
}]);
```

## SWIFT

```
let properties = SDLCloudAppProperties(appID: <#app id#>)
properties.authToken = <#auth token#>
let setCloud = SDLSetCloudAppProperties(properties: properties)
sdIManager.send(request: setCloud) { (req, res, err) in
    guard let response = res as? SDLSetCloudAppPropertiesResponse,
    response.success.boolValue == true, err == nil else {
        return
    }

    <#Use the response#>
}
```

## Getting Cloud App Properties

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

## OBJECTIVE-C

```
SDLGetCloudAppProperties *getCloud = [[SDLGetCloudAppProperties
alloc] initWithAppID:<#app id#>];
[self.sdIManager sendRequest:getCloud withResponseHandler:^(
__typeof(SDLRPCRequest * _Nullable request, __typeof(SDLRPCResponse
* _Nullable response, NSError * _Nullable error) {
    if (!response || !response.success.boolValue) {
        SDLLogE(@"Error sending set cloud properties: Req %@, Res %@,
err %@", request, response, error);
        return;
    }

    SDLGetCloudAppPropertiesResponse *setCloudResponse = (
SDLGetCloudAppPropertiesResponse *)response;
    <#Use the response#>
}]);
```

## SWIFT

```
let getCloud = SDLGetCloudAppProperties(appID: <#app id#>)  
sdIManager.send(request: getCloud) { (req, res, err) in  
    guard let response = res as? SDLGetCloudAppPropertiesResponse,  
          response.success.boolValue == true, err == nil else {  
        return  
    }  
  
    <#Use the response#>  
}
```

---

## GETTING THE CLOUD APP ICON

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

# Getting the Authentication Token

When users install cloud apps from an OEM's app store, they may be asked to login to that cloud app using the app store. After logging in, app store can save the `authToken` in the local policy table to be used later by the cloud app for user authentication.

A cloud app can retrieve its `authToken` from local policy table after starting the RPC service. The `authToken` can be used later by the app to authenticate the user:

## OBJECTIVE-C

```
NSString *authToken = self.sdIManager.authToken;
```

```
let authToken = sdlManager.authToken
```

## Getting CloudAppVehicleID (Optional)

The `CloudAppVehicleID` is an optional parameter used by cloud apps to identify a head unit. The content of `CloudAppVehicleID` is up to the OEM's implementation. Possible values could be the VIN or a hashed VIN.

The `CloudAppVehicleID` value can be retrieved as part of the `GetVehicleData` RPC. To find out more about how to retrieve `CloudAppVehicleID`, check out the [Retrieving Vehicle Data](#) section.

## Introduction

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

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

- Navigation apps don't use base screen templates. Their main view is the video stream sent from the device.
- Navigation apps can send audio via a binary stream. This will attenuate the current audio source and should be used for navigation commands.
- Navigation apps can receive touch events from the video stream.

# Configuring a Navigation App

The basic connection setup is similar for all apps. Please follow the [Integration Basics](#) guide for more information.

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

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

## OBJECTIVE-C

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

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())
```

### MUST

When compiling your app for production, 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 phone is backgrounded or when the phone is sleeping/locked. Text will automatically be displayed telling the user that they must bring the application to the foreground.

## Transports for Video Streaming

Transports are automatically handled for you. As of SDL v6.1, the iOS library will automatically manage primary transports and secondary transports for video streaming. If Wi-Fi is available, the app will automatically connect using it *after* connecting over USB / Bluetooth. This is the only way that Wi-Fi will be used in a production setting.

## CarWindow

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

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

## MUST

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

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

1. Choose how `CarWindow` captures and renders the screen using the `carWindowRenderingType` enum.
2. By default, when using `CarWindow`, the `SDLTouchManager` will sync its touch updates to the framerate. To disable this feature, set `SDLTouchManager.enableSyncedPanning` to `NO`.
3. `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 `sdlManager.streamManager.rootViewController` to the new view controller. This will also update the [haptic parser](#).

## Mirroring the Device Screen vs. Off-Screen UI

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

---

### OFF-SCREEN

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

## OBJECTIVE-C

```
UIViewController *offScreenViewController = <#Acquire a  
UIViewController#>;  
self.sdIManager.streamManager.rootViewController =  
offScreenViewController;
```

## SWIFT

```
let offScreenViewController = <#Acquire a UIViewController#>  
sdIManager.streamManager?.rootViewController =  
offScreenViewController
```

---

## MIRRORING THE DEVICE SCREEN

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

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

frame after a notification from the latter can also cause weird behavior when setting the new frame.

4. Configure your SDL app so the lock screen is **always visible**. If you do not do this, video streaming can stop when the device is rotated.

## Showing a New View Controller

Simply update the streaming media manager's `rootViewController` to the new view controller. This will also automatically update the **haptic parser**.

## 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

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

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

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

## Audio Stream Lifecycle

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

# Audio Stream Manager

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

---

## PLAYING FROM FILE

### OBJECTIVE-C

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

### SWIFT

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

---

## PLAYING FROM DATA

### OBJECTIVE-C

```
[self.sdlManager.streamManager.audioManager pushWithData:  
audioData];  
[self.sdlManager.streamManager.audioManager playNextWhenReady];
```

## SWIFT

```
sdIManager.streamManager?.audioManager.push(withData: audioData)
sdIManager.streamManager?.audioManager.playNextWhenReady()
```

---

## IMPLEMENTING THE DELEGATE

### OBJECTIVE-C

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

- (void)audioStreamManager:(SDLAudioStreamManager *)
audioManager errorDidOccurForDataBuffer:(NSError *)error {
}

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

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

## SWIFT

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

}

func audioStreamManager(_ audioManager: SDLAudioStreamManager,
errorDidOccurForDataBuffer error: Error) {

}

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

func audioStreamManager(_ audioManager: SDLAudioStreamManager,
dataBufferDidFinishPlayingSuccessfully 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. If successful playback will begin immediately.

## OBJECTIVE-C

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

if (![self.sdlManager.streamManager sendAudioData:audioData]) {
    <#Could not send audio data#>
}
```

## SWIFT

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

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

if !streamManager.sendAudioData(audioData) {
    <#Could not send audio data#>
}
```

# Touch Input

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

## NOTE

You must have a valid and approved `appld` from an OEM in order to receive touch events.

## Using SDLTouchManager

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

### OBJECTIVE-C

```
self.sdIManager.streamManager.touchManager.touchEventDelegate =  
self
```

### SWIFT

```
sdIManager.streamManager.touchManager.touchEventDelegate = self
```

## 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

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

```
// sdl_ios v6.3+
[self.sdlManager subscribeToRPC:SDLDidReceiveTouchEventNotification
withObserver:self selector:@selector(touchEventAvailable:)];

// Pre sdl_ios v6.3
[[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

```
// sdl_ios v6.3+
sdlManager.subscribe(to: .SDLDidReceiveTouchEvent, observer: self,
selector: #selector(touchEventAvailable(_:)))

// Pre sdl_ios v6.3
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 views on the screen are touchable and focusing / highlighting on those areas when the user moves the trackpad or click wheel. When the user selects within a view, the center of that area will be "touched".

## NOTE

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

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

# Automatic Focusable Rects

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

## NOTE

This is only supported on iOS 9 devices and above. If you want to support this feature 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 UI 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 `UIView`s or OpenGL views), then you will have to manually send and update the focusable rects using `SDLSendHapticData`. This request, when

send 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

```
SDLRectangle *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

```
guard let viewRect = SDLRectangle(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.

When your navigation app is guiding the user to a specific destination, you can provide the user with visual and audio turn-by-turn prompts. These prompts will be presented even when your SDL app is backgrounded or a phone call is ongoing.

While your app is navigating the user, you will also want to send turn by turn directions. This is useful if your app is in the background or if the user is in the middle of a phone call, and gives the system additional information about the next maneuver the user must make.

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

## NOTE

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

## Visual Turn Directions

The visual data is sent using the `SDLShowConstantTBT` RPC. The main properties that should be set are `navigationText1`, `navigationText2`, and `turnIcon`. A best practice for navigation apps is to use the `navigationText1` as the

direction to give (i.e. turn right) and `navigationText2` to provide the distance to that direction (i.e. 3 mi.).

## Audio Turn Directions

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

# Sending Both Audio and Visual Turn Directions

## 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]) {
        <#Error sending ShowConstantTBT#>
        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]) {
            <#Error sending AlertManeuver#>
            return;
        }

        <#Both ShowConstantTBT and AlertManeuver were sent
successfully#>
    }];
}];
```

## SWIFT

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

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

sdIManager.send(request: turnByTurn) { (request, response, error) in
    guard response?.resultCode == .success else {
        <#Error sending ShowConstantTBT#>
        return
    }

    let alertManeuver = SDLAlertManeuver(tts: "In 3 miles turn right",
softButtons: nil)
    self.sdIManager.send(request: alertManeuver, responseHandler: { (
request, response, error) in
        guard response?.resultCode == .success else {
            <#Error sending AlertManeuver#>
            return
        }

        <#Both ShowConstantTBT and AlertManeuver were sent
successfully#>
    })
}
```

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

## Clearing the Turn Directions

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

## OBJECTIVE-C

```
SDLShowConstantTBT* clearTurnByTurn = [[SDLShowConstantTBT alloc]
init];
clearTurnByTurn.maneuverComplete = @YES;

[self.sdlManager sendRequest:clearTurnByTurn withResponseHandler:^(
SDLRPCRequest *request, SDLRPCResponse *response, NSError *error)
{
    if (![response.resultCode isEqualToEnum:SDLResultSuccess]) {
        <#Error sending TBT#>
        return;
    }

    <#TBT successfully cleared#>
}];
```

## SWIFT

```
let clearTurnByTurn = SDLShowConstantTBT()
clearTurnByTurn.maneuverComplete = true as NSNumber

sdlManager.send(request: clearTurnByTurn) { (request, response, error)
in
    guard response?.resultCode == .success else {
        <#Error sending TBT#>
        return
    }

    <#TBT successfully cleared#>
}
```

# Configuring SDL Logging

Starting at SDL iOS v5.0 and above, a new powerful built-in logging framework was 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 [01F/539] (SDL)Protocol - I'm a log!
```

### Default:

```
09:52:07:324 [01F/539] (SDL)Protocol:SDLV2ProtocolHeader:25 - I'm also a log!
```

### Detailed:

```
09:52:07:324 [01F/539] 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 (default 3) which will be stored on the device, specifically in the `Documents/smartdevicelink/log/` folder. The file names will be timestamped with the start time.

To access the file, you can either access it from runtime on the device (for example, to attach it to an email that the user sends), or if you have access to the device, you can access them via iTunes as well with small modifications to your app:

1. Add the key `UIFileSharingEnabled` to your `info.plist`. Set the value to `YES`.
2. Connect the device to a computer that has iTunes installed.
3. Open iTunes, click on the icon for the device, then click on "File Sharing" > "(Your App Name)"

4. You should see a folder called "smartdevicelink". Select the file and click save. When you open the folder on your computer, you will see the log files for each session (default maxes out at 3).

You should not keep this info.plist key when you submit your app to Apple.

---

## 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 `SDLLogFileModule` class and add it to the configuration. Modules are used when outputting a log message. The log message may specify a module instead of a

specific file name for clarity's sake. The SDL library will automatically add the modules corresponding to its own files after you submit your configuration. For your specific use case, you may wish to provide a module corresponding to your whole app's integration and simply name it with your app's name, or, you could split it up further if desired. To add modules to the configuration:

## OBJECTIVE-C

```
logConfig.modules = [logConfig.modules setByAddingObjectsFromArray
:@[[SDLLogFileModule moduleName:@"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(byDisallowingString: "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.

## USB Connection

Status: **Disconnected**

## EASession

Status: **Closed**

## Server



Status: **Not Started**

Address:

Port:

## TCP/IP

Status: **Disconnected**

*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.

## USB Connection

Status: **Connected**

## EASession

Status: **Connecting...**

## Server



Status: **Not Started**

Address:

Port:

## TCP/IP

Status: **Disconnected**

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

## USB Connection

Status: **Connected**

## EASession

Status: **Connected**

## Server



Status: **Not Started**

Address:

Port:

## TCP/IP

Status: **Disconnected**

*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**.

## USB Connection

Status: Connected

## EASession

Status: Connected

## Server



Status: Available

Address: 1.2.3.4

Port: 2776

## TCP/IP

Status: Disconnected

*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*.

## USB Connection

Status: Connected

## EASession

Status: Connected

## Server



Status: Available

Address: 1.2.3.4

Port: 2776

## TCP/IP

Status: 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*.

## USB Connection

Status: Connected

## EASession

Status: Connected

## Server



Status: Available

Address: 1.2.3.4

Port: 2776

## TCP/IP

Status: 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**

## Settings

[Next](#)

# Spec

## File

Default\_Spec.xml



# SmartDeviceLink Proxy

## Connection Type

TCP

## IP Address

127.0.0.1

## Port

12345

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**

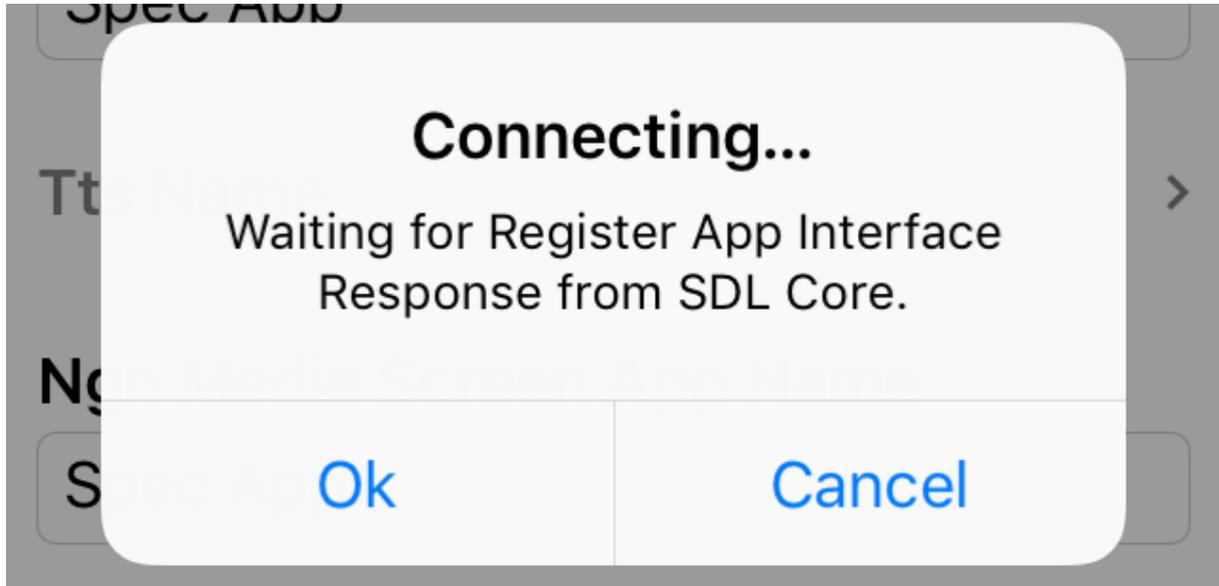


**Is Media Application**



**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

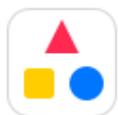
## RPC Builder



AddCommand



AddSubMenu



Alert



AlertManeuver



ChangeRegistration



CreateInteractionChoiceSet



DeleteCommand



DeleteFile



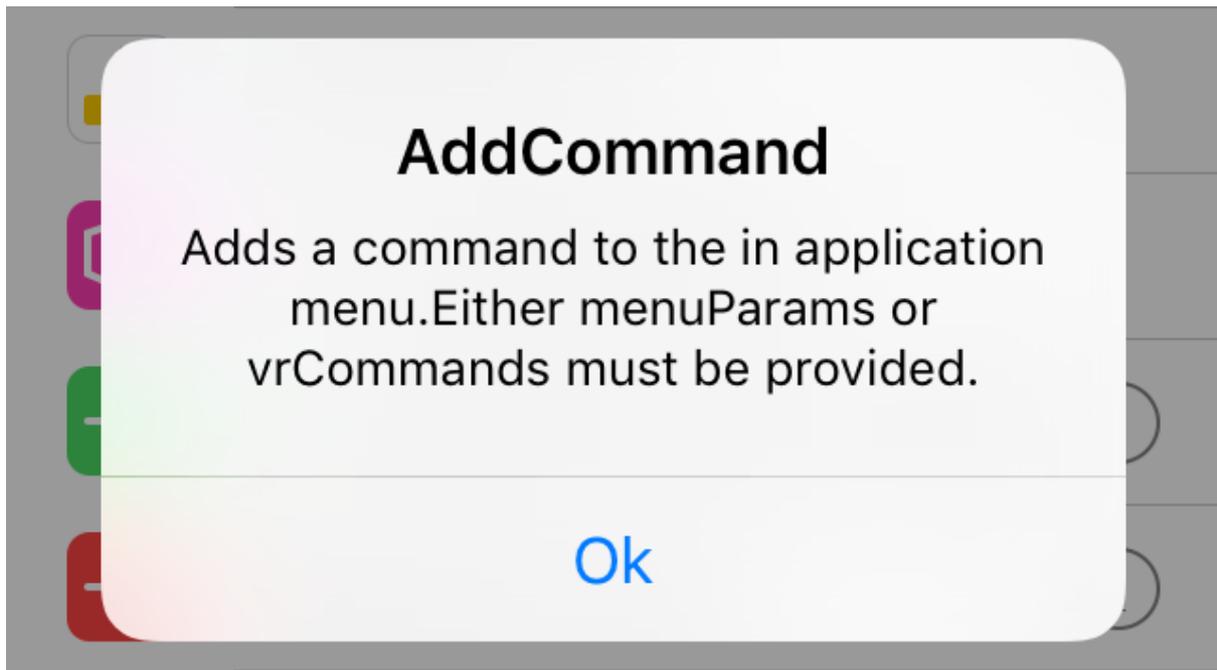
DeleteInteractionChoiceSet



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.

[< RPC Builder](#)[Show](#)[Send](#)

Tap Parameter Name to Include/Remove from RPC.

### Main Field 1

### Main Field 2

### Main Field 3

### Main Field 4

### Alignment

\* 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.

## Main Field 1

500 characters

## Main Field 2

500 characters

## 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.



---

## 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 `moduleName`/`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], //
            [RBNewModuleViewController classString] // Module
        ];
    }
    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.

## 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] OnHMISStatus (notification)

11:16:26.15

[2] OnHMISStatus (notification)

11:16:28.43

[1] Alert (request)

11:16:37.17

[3] OnHMISStatus (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.



## [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"
  );
}
```

## 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) NSInteger correlationID;

@property (nonatomic, strong) NSNumber *appId;

@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.remotImages removeAllObjects];
    self.vehicleDataSubscribed = NO;
    self.applconId = nil;

    // Notify the app delegate to clear the lockscreen
    [self hsdI_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 hsdI_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 hsdI_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 hsdI_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 hsdI_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 hsdI_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:

- hsdI\_uploadImages
- onListFilesResponse:
- onPutFileResponse:
- hsdI\_setAppIcon

And can also remove the `remotelmages` 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 `applied` 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 appld:Appld];

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

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

        UIImage* applcon = [UIImage imageNamed:IconFile];
        if (applcon) {
            _lifecycleConfiguration.applcon = [SDLArtwork
artworkWithImage:applcon 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 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 its 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 `HS` `DLProxyManager`:

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 `SDLProxyListerner`, 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)hndl_addCommands {
    NSLog(@"hndl_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 hndl_getNextCorrelationId];
        [self.proxy sendRPC:show];

        SDLSpeak *speak = [SDLRPCRequestFactory buildSpeakWithTTS:
@"Test Command" correlationID:[self hndl_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)hndl_addCommands {
    NSLog(@"hndl_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)hmdl_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)hmdl_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 `hmdl_uploadImage:withCorrelationID:` from

```

- (void)hndl_uploadImage:(NSString *)imageName withCorrelationID:(
NSNumber *)corrId {
    NSLog(@"hndl_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 numberWithIntUnsignedLong:
pngData.length];
                putFile.bulkData = pngData;
                putFile.correlationID = corrId;
                [self.proxy sendRPC:putFile];
            }
        }
    }
}

```

to

```

- (void)hndl_uploadImage:(NSString *)imageName {
    NSLog(@"hndl_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
sdFileManager.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
sdFileManager.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;
```

## Can I Integrate SDL into a React Native App?

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

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

To install SDL into your React Native app, you will need to follow [this guide](#) to integrate the SDL library into your application using React Native's Native Modules feature. You must make sure you have [Native Modules](#) installed as a dependency in order to use 3rd party APIs in a React Native application. If this is not done your app will not work with SmartDeviceLink. Native API methods are not exposed to JavaScript automatically, this must be done manually by you.

## NOTE

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

# Integration Basics

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

## Creating the RCTBridge

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

OBJECTIVE-C

## ProxyManager.h

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

@interface ProxyManager : NSObject <RCTBridgeModule>

<#Proxy Manager code#>

@end
```

## ProxyManager.m

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

```
@implementation ProxyManager

RCT_EXPORT_MODULE();
<#Proxy Manager code#>

@end
```

## SWIFT

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

```
@objc(ProxyManager)

class ProxyManager: NSObject {

<#Proxy Manager Code#>

}
```

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

```
#import "React/RCTBridgeModule.h"

@interface RCT_EXTERN_MODULE(ProxyManager, NSObject)

@end
```

## Emitting Event Notifications to JavaScript

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

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

### OBJECTIVE-C

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

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

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

## SWIFT

```
let softButton = SDLSoftButtonObject(name: "Button", state:
SDLSoftButtonState(stateName: "State", text: "Data", artwork: nil),
handler: { (buttonPress, butonEvent) in
    guard buttonPress == nil else { return }

    let userInfo = ["sdIManager": self.sdIManager]
    NotificationCenter.default.post(name: NSNotification.Name(rawValue:
<#Notification Name#>), object: nil, userInfo: managers)
})

self.sdIManager.screenManager.softButtonObjects = [softButton];
```

---

## CREATE THE EVENTEMITTER BRIDGE CLASS

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

### OBJECTIVE-C

SDLEventEmitter.h

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

NS_ASSUME_NONNULL_BEGIN

@interface SDLEventEmitter : RCTEventEmitter

@end

NS_ASSUME_NONNULL_END
```

## SDLEventEmitter.m

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

@implementation SDLEventEmitter

RCT_EXPORT_MODULE()

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

    return self;
}

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

// Run this code when the subscribed event notification is received
- (void)getDoActionNotification:(NSNotification *)notification {
    if(self.sdlManager == nil) {
        self.sdlManager = notification.userInfo[@"sdlManager"];
    }

    // Send the event to your React Native code with a dictionary of
    information
    [self sendEventWithName:@"DoAction" body:@{@"type":
@"actionType"}];
}

@end
```

## SWIFT

```
@objc(SDLEventEmitter)
class SDLEventEmitter: RCTEventEmitter {

    override init() {
        // Subscribe to event notifications sent from ProxyManager
        NotificationCenter.default.addObserver(self, selector: #selector(
doAction(_:)), name: Notification.Name(rawValue: "<#Notification
Name#>", object: nil)
        super.init()
    }

    // Required Method defining known action names
    override func supportedEvents() -> [String]! {
        return ["DoAction"]
    }

    // Run this code when the subscribed event notification is received
    @objc func doAction(_ notification: Notification) {
        if self.sdManager == nil {
            self.sdManager = notification.userInfo["sdManager"]
        }

        // Send the event to your React Native code with a dictionary of
information
        sendEvent(withName: "DoAction", body: ["type": "actionType"])
    }
}
```

## JAVASCRIPT

The above example will call into your JavaScript code with an event type `DoAction`. Inside your React Native (JavaScript) code, create a `NativeEventEmitter` object within your `EventEmitter` module and add a listener for the event.

```

import { NativeEventEmitter, NativeModules } from 'react-native';
const { SDLEventEmitter } = NativeModules;

const testEventEmitter = new NativeEventEmitter(SDLEventEmitter);

// Build a listener to listen for events
const testData = testEventEmitter.addListener(
  'DoAction',
  () => SDLEventEmitter.eventCall({
    "data": {
      "low": "77",
      "high": "87",
      "currentTemp": "82",
      "rain": "50%"
    }
  })
)
)

```

## Exposing Native Methods to JavaScript

The last step is to wrap any native code methods you wish to expose to your JavaScript code inside `RCT_EXPORT_METHOD` for Objective-C and `RCT_EXTERN_METHOD` for Swift. We've seen above how native code can send notifications to your JavaScript code, now we will see how your JavaScript code can send notifications into your native SmartDeviceLink code. Inside the `SDLEventEmitter.m` file add the following method:

## OBJECTIVE-C

```
RCT_EXPORT_METHOD(eventCall:(NSDictionary *)dict) {
    [self.sdlManager.screenManager beginUpdates];

    self.sdlManager.screenManager.textField1 = [NSString
stringWithFormat:@"Low: %@ %F", [RCTConvert NSString:dict[@"data"]
[@"low"]]];
    self.sdlManager.screenManager.textField2 = [NSString
stringWithFormat:@"High: %@ %F", [RCTConvert NSString:dict[@"data"]
[@"high"]]];

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

## SWIFT

If you're making a React Native application and using native Swift code, you will need to create the Objective-C bridger for the `SDLEventEmitter` class you created above. Wrap the method(s) you wish to expose in a `RCT_EXTERN_METHOD` macro inside your wrapper class. This wrapper will allow the JavaScript code to talk with your native code.

### NOTE

Make sure you add `#import "React/RCTEventEmitter.h"` to the apps bridging header.

```

#import "React/RCTBridgeModule.h"
#import "React/RCTEventEmitter.h"

@interface RCT_EXTERN_MODULE(SDLEventEmitter, RCTEventEmitter)

RCT_EXTERN_METHOD(eventCall:(eventCall: (id)dict))

@end

```

Add the following method to `SDLEventEmitter.swift`:

```

@objc func eventCall(_ dict: NSDictionary) {
    self.sdlManager.screenManager.beginUpdates()
    let data = dict["data"]! as! NSDictionary
    self.sdlManager.screenManager.textField1 = "Low: \(data["low"]!) °F"
    self.sdlManager.screenManager.textField2 = "High: \(data["high"]!) °F"
    self.sdlManager.screenManager.endUpdates()
}

```

By now you should have a basic React Native application that can send a message from the Native side to the React Native layer. If done correctly the application should update the SDL UI when clicking the soft button on the head unit. The above documentation walked you through how to send a message to React Native and receive a message containing data back.