sdl

JavaSE Guides

Document current as of 04/02/2020 12:18 PM.

Installation

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

NOTE

The SDL SDK is currently supported on Java 7 (1.7) and above.

Install SDL SDK

Each SDL JavaSE library release is published to JCenter. By adding a few lines in their app's gradle script, developers can compile with the latest SDL JavaSE release.

To gain access to the JCenter repository, make sure your app's `build.gradle` file includes the following:

```
repositories {
    google()
    jcenter()
}
```

Gradle Build

To compile with the a release of SDL JavaSE, include the following line in your app's `build.gradle` file,

```
dependencies {
    implementation 'com.smartdevicelink:sdl_java_se:{version}'
}
```

and replace `{version}` with the desired release version in format of `x.x.x`. The list of releases can be found here.

Examples

To compile release 4.10.1, use the following line:

```
dependencies {
    implementation 'com.smartdevicelink:sdl_java_se:4.10.1'
}
```

To compile the latest minor release of major version 4, use:

```
dependencies {
    implementation 'com.smartdevicelink:sdl_java_se:4.+'
}
```

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.

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.

In this guide, we exclusively use IntelliJ. We are going to set-up a bare-bones application so you get started using SDL.

NOTE

The SDL Java library supports Java 7 and above.

SmartDeviceLink Service

A SmartDeviceLink Service should be created to manage the lifecycle of the SDL session. The `SdlService` should build and start an instance of the `SdlManager` which will automatically connect with a head unit when available. This `SdlManager` will handle sending and receiving messages to and from SDL after it is connected.

```
public class SdlService {
    //...
}
```

Implementing SDL Manager

In order to correctly connect to an SDL enabled head unit developers need to implement methods for the proper creation and disposing of an `SdlManager` in our `SdlService`.

NOTE

An instance of SdlManager cannot be reused after it is closed and properly disposed of. Instead, a new instance must be created. Only one instance of SdlManager should be in use at any given time.

```
public class SdlService {

    //The manager handles communication between the application and SDL
    private SdlManager sdlManager = null;

    public SdlService(BaseTransportConfig config){
        buildSdlManager(config);
    }

    public void start() {
        if(sdlManager != null){
            sdlManager.start();
        }
    }

    public void stop() {
        if (sdlManager != null) {
            sdlManager.dispose();
            sdlManager = null;
        }
    }

    //...

    private void buildSdlManager(BaseTransportConfig transport) {

        if (sdlManager == null) {

            // The app type to be used
            Vector<AppHMIType> appType = new Vector<>();
            appType.add(AppHMIType.MEDIA);

            // The manager listener helps you know when certain events that pertain
to the SDL Manager happen
            SdlManagerListener listener = new SdlManagerListener() {

                @Override
                public void onStart(SdlManager sdlManager) {
                    // After this callback is triggered the SdlManager can be used to
interact with the connected SDL session (updating the display, sending RPCs, etc)
                }

                @Override
                public void onDestroy(SdlManager sdlManager) {
                }

                @Override
                public void onError(SdlManager sdlManager, String info, Exception e) {
                }
            };
            // Create App Icon, this is set in the SdlManager builder
            SdlArtwork appIcon = new SdlArtwork(ICON_FILENAME,
FileType.GRAPHIC_PNG, ICON_PATH, true);

            // The manager builder sets options for your session
            SdlManager.Builder builder = new SdlManager.Builder(APP_ID, APP_NAME,
listener);
            builder.setAppTypes(appType);
            builder.setTransportType(transport);
            builder.setAppIcon(appIcon);
            sdlManager = builder.build();
            sdlManager.start();
        }

    }
}
```

NOTE

The `sdlManager` must be shutdown properly if this class is shutting down in the respective method using the method `sdlManager.dispose()`.

Determining SDL Support

You have the ability to determine a minimum SDL protocol and a 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.

```
builder.setMinimumProtocolVersion(new Version("3.0.0"));
builder.setMinimumRPCVersion(new Version("4.0.0"));
```

Listening for RPC notifications and events

We can listen for specific events using `SdlManager`'s builder `setRPCNotificationListeners`. The following example shows how to listen for HMI Status notifications. Additional listeners can be added for specific RPCs by using their corresponding `FunctionID` in place of the `ON_HMI_STATUS` in the following example and casting the `RPCNotification` object to the correct type.

EXAMPLE OF A LISTENER FOR HMI STATUS:

```
Map<FunctionID, OnRPCNotificationListener> onRPCNotificationListenerMap =
new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus onHMIStatus = (OnHMIStatus) notification;
        if (onHMIStatus.getHmiLevel() == HMILevel.HMI_FULL &&
onHMIStatus.getFirstRun()){
            // first time in HMI Full
        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

Main Class

Now that the basic connection infrastructure is in place, we should add methods to start the `SdlService` when our application starts. In `main(String[] args)` in your main class, you will create and start an instance of the `SdlService` class.

You will also need to fill in what port the app should listen on for an incoming web socket connection.

```
public class Main {

    Thread thread;
    SdlService sdlService;

    public static void main(String[] args) {
        Main main = new Main();
        main.startSdlService();

    }

private void startSdlService() {

        thread = new Thread(new Runnable() {

            @Override
            public void run() {
                sdlService  = new SdlService(new WebSocketServerConfig(PORT, -1));
                sdlService.start();


            }
        });
        thread.start();
    }
}
```

Connecting to an Infotainment System

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

Your SDL embedded app will only work with head units that support RPC Spec v5.1+.

Configuring the Connection

Generic SDL Core

To connect to your app to a local Ubuntu SDL Core-based emulator you need to know the IP address of the machine that is running the cloud app. If needed, running `ifconfig` in the terminal will give you the current network configuration information.

POLICY TABLE CONFIGURATION

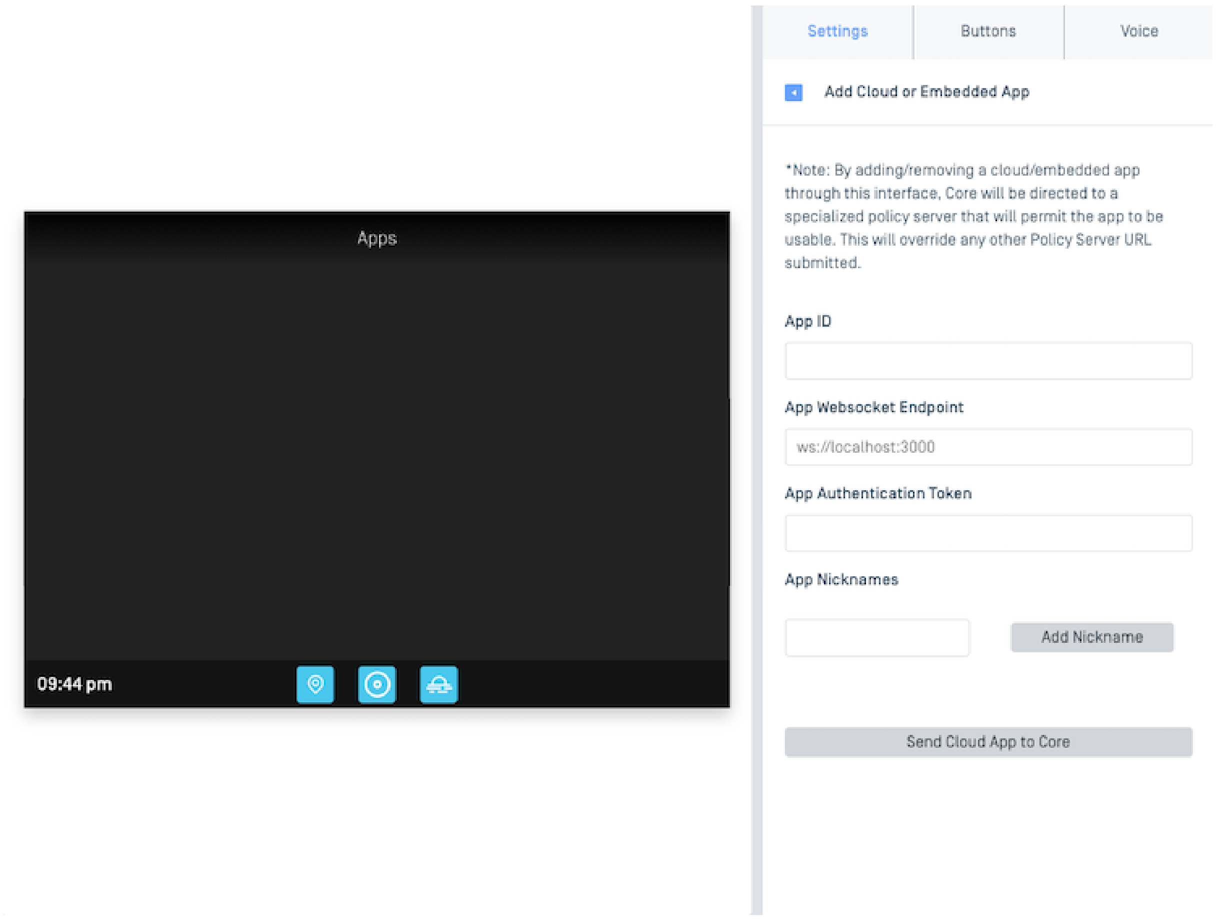
Once you know the IP address, you need to set the websocket `endpoint` and app `nicknames` for your SDL app in the policy table under the "app_policies" section. This will let Core know where your instance of the SDL app is running. The websocket endpoint needs to include both the IP address and port: `ws://<ip address>:<port>/`.

```
"<Your SDL App ID>": {
    "keep_context": false,
    "steal_focus": false,
    "priority": "NONE",
    "default_hmi": "NONE",
    "groups": ["Base-4"],
    "RequestType": [],
    "RequestSubType": [],
    "hybrid_app_preference": "CLOUD",
    "endpoint": "ws://<ip address>:<port>",
    "enabled": true,
    "auth_token": "",
    "cloud_transport_type": "WS",
    "nicknames": ["<app name>"]
}
```

For more information about policy tables please visit the Policy Table guide.

Manticore

If you are using Manticore, the app connection information can be easily added in the settings tab of the Manticore web page. Please note that Manticore needs to access your machine's IP address in order to be able to start a websocket connection with your app. If you are hosting the app on your local machine, you may need to do extra setup to make your machine publicly accessible.



*Note: By adding/removing a cloud/embedded app through this interface, Core will be directed to a specialized policy server that will permit the app to be usable. This will override any other Policy Server URL submitted.

App ID

App Websocket Endpoint

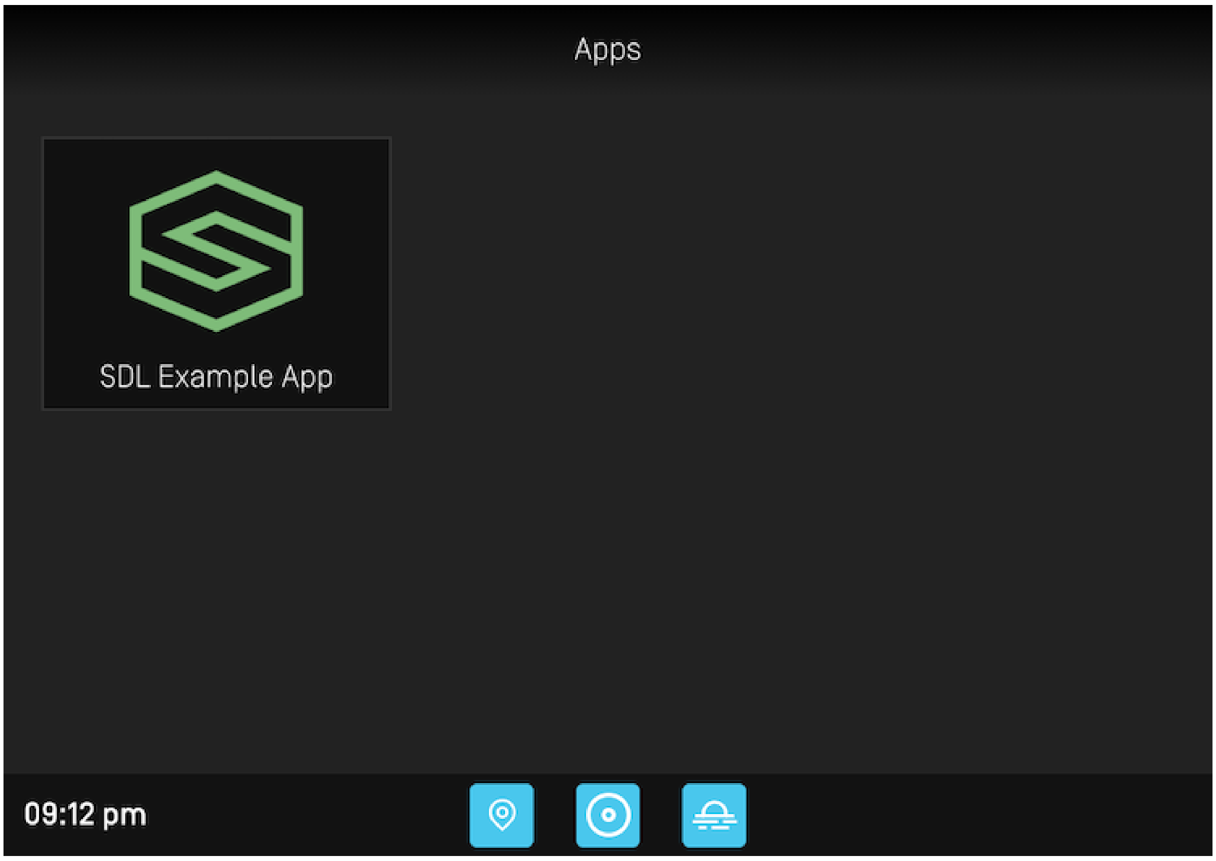
ws://localhost:3000

App Authentication Token

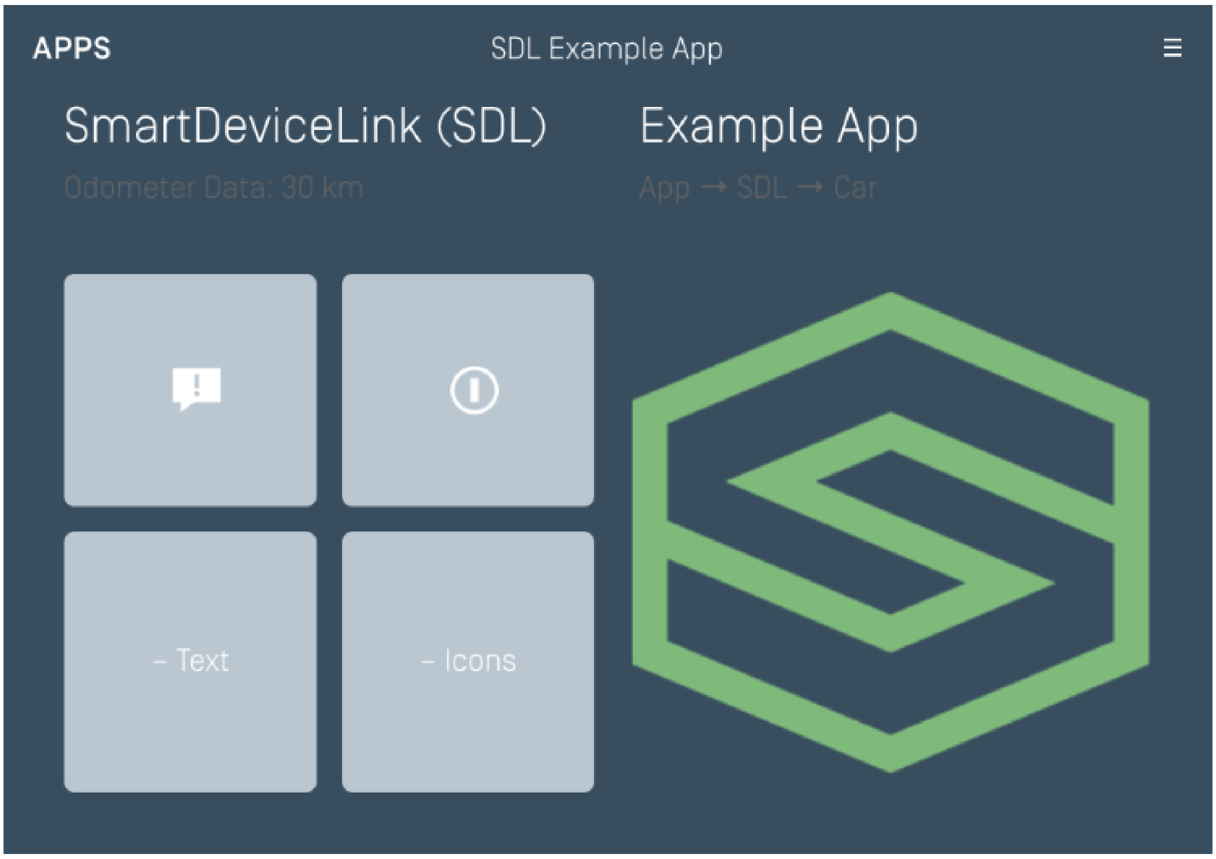
App Nicknames

Running the SDL App

Once you have a configured instance of Core running, you should see your SDL app name appear in a box on HMI. However, nothing will happen when you tap on the box until you build and run your SDL app.



Once your SDL app is running, either locally in an IDE or on a server, you will be able to launch the SDL app by clicking on the app icon in the HMI.



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

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 `Builder`. 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 `Builder`. 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 `sdlManager.getRegisterAppInterfaceResponse()` 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.

```
Language headUnitLanguage =
sdlManager.getRegisterAppInterfaceResponse().getLanguage();
Language headUnitHMILanguage =
sdlManager.getRegisterAppInterfaceResponse().getHmiDisplayLanguage();
```

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 `Builder`.
2. Implement the `sdlManagerListener`'s `managerShouldUpdateLifecycle` 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 `LifecycleConfigurationUpdate` with the new `appName` and/or `ttsName`.

```
@Override
public LifecycleConfigurationUpdate managerShouldUpdateLifecycle(Language
language){
    String appName;
    switch (language) {
        case ES_MX:
            appName = APP_NAME_ES;
            break;
        case FR_CA:
            appName = APP_NAME_FR;
            break;
        default:
            return null;
    }

    return new
LifecycleConfigurationUpdate(appName,null,TTSChunkFactory.createSimpleTTSC
 null);
}
```

Understanding Permissions

While creating your SDL app, remember that just because your app is connected to a head unit it does not mean that the app has permission to send the RPCs you want. If your app does not have the required permissions, requests will be rejected. There are three important things to remember in regards to permissions:

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

HMI Levels

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

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

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

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

Monitoring the HMI Level

Monitoring HMI Status is possible through an `OnHMIStatus` notification that you can subscribe to via the `SdlManager` 's `addOnRPCNotificationListener` .

```
Map<FunctionID, OnRPCNotificationListener> onRPCNotificationListenerMap =
new HashMap<>();
onRPCNotificationListenerMap.put(FunctionID.ON_HMI_STATUS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnHMIStatus onHMIStatus = (OnHMIStatus) notification;
        if (onHMIStatus.getHmiLevel() == HMILevel.HMI_FULL &&
onHMIStatus.getFirstRun()){
            // first time in HMI Full
        }
    }
});
builder.setRPCNotificationListeners(onRPCNotificationListenerMap);
```

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

```
boolean allowed =
sdlManager.getPermissionManager().isRPCAllowed(FunctionID.SHOW);

// You can also check if a permission parameter is allowed
boolean parameterAllowed =
sdlManager.getPermissionManager().isPermissionParameterAllowed(FunctionID.
 GetVehicleData.KEY_RPM);
```

Checking Current Permissions of a Group of RPCs

```
List<PermissionElement> permissionElements = new ArrayList<>();
permissionElements.add(new PermissionElement(FunctionID.SHOW, null));
permissionElements.add(new
PermissionElement(FunctionID.GET_VEHICLE_DATA,
Arrays.asList(GetVehicleData.KEY_RPM, GetVehicleData.KEY_SPEED)));

int groupStatus =
sdlManager.getPermissionManager().getGroupStatusOfPermissions(permissionE


switch (groupStatus) {
    case PermissionManager.PERMISSION_GROUP_STATUS_ALLOWED:
        // Every permission in the group is currently allowed
        break;
    case PermissionManager.PERMISSION_GROUP_STATUS_DISALLOWED:
        // Every permission in the group is currently disallowed
        break;
    case PermissionManager.PERMISSION_GROUP_STATUS_MIXED:
        // Some permissions in the group are allowed and some disallowed
        break;
    case PermissionManager.PERMISSION_GROUP_STATUS_UNKNOWN:
        // The current status of the group is unknown
        break;
}
```

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

```
List<PermissionElement> permissionElements = new ArrayList<>();
permissionElements.add(new PermissionElement(FunctionID.SHOW, null));
permissionElements.add(new
PermissionElement(FunctionID.GET_VEHICLE_DATA,
Arrays.asList(GetVehicleData.KEY_RPM,
GetVehicleData.KEY_AIRBAG_STATUS)));

Map<FunctionID, PermissionStatus> status =
sdlManager.getPermissionManager().getStatusOfPermissions(permissionElemen


if (status.get(FunctionID.GET_VEHICLE_DATA).getIsRPCAllowed()){
    // GetVehicleData RPC is allowed
}

if
(status.get(FunctionID.GET_VEHICLE_DATA).getAllowedParameters().get(GetVeh
{
    // rpm parameter in GetVehicleData RPC is allowed
}
```

Observing Permissions

If desired, you can set a listener for a group of permissions. The listener 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 `PERMISSION_GROUP_TYPE_ANY`. If you only want to be notified when all of the RPCs in the group are allowed, set the `groupType` to `PERMISSION_GROUP_TYPE_ALL_ALLOWED`.

```
List<PermissionElement> permissionElements = new ArrayList<>();
permissionElements.add(new PermissionElement(FunctionID.SHOW, null));
permissionElements.add(new
PermissionElement(FunctionID.GET_VEHICLE_DATA,
Arrays.asList(GetVehicleData.KEY_RPM,
GetVehicleData.KEY_AIRBAG_STATUS)));


UUID listenerId =
sdlManager.getPermissionManager().addListener(permissionElements,
PermissionManager.PERMISSION_GROUP_TYPE_ANY, new
OnPermissionChangeListener() {
    @Override
    public void onPermissionsChange(@NonNull Map<FunctionID,
PermissionStatus> allowedPermissions, @NonNull int permissionGroupStatus) {
        if
(allowedPermissions.get(FunctionID.GET_VEHICLE_DATA).getIsRPCAllowed()) {
            // GetVehicleData RPC is allowed
        }

        if
(allowedPermissions.get(FunctionID.GET_VEHICLE_DATA).getAllowedParameters
{
            // rpm parameter in GetVehicleData RPC is allowed
        }
    }
});
```

Stopping Observation of Permissions

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

```
sdlManager.getPermissionManager().removeListener(listenerId);
```

Additional HMI State Information

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

Audio Streaming State

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

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

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

```
@Override
public void onNotified(RPCNotification notification) {
    OnHMIStatus status = (OnHMIStatus) notification;
    AudioStreamingState streamingState = notification.getAudioStreamingState();
}
```

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.

```
@Override
public void onNotified(RPCNotification notification) {
    OnHMIStatus status = (OnHMIStatus) notification;
    SystemContext systemContext = notification.getSystemContext();
}
```

Checking Supported Features

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

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

Checking the System Capability Manager

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

Handling RPC Error Responses

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

```
<#Your Request#>.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (!response.getSuccess()) {
            <#The request was not successful, check the onError handler for more 
information#>
            return;
        }

        <#The request was successful#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        <#The request was not successful, check the resultCode and info for more 
information#>
    }
});
sdlManager.sendRPC(<#Your Request#>);
```

Checking if a Feature is Supported by Version

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

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

Example Apps

This guide takes you through the steps needed to get the sample project, *Hello Sdl*, connected a module.

First, make sure you download or clone the latest release from GitHub. It is a project within the SDL Java Suite root directory. Then, open the *Hello Sdl* project in IntelliJ IDEA.

Connecting to an Infotainment System

To connect the sample app to the infotainment system, please follow the instructions in the Connecting to an Infotainment System guide.

Adaptive Interface Capabilities

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

You can access these properties on the `sdlManager.getSystemCapabilityManager` instance.

System Capability Manager Properties

PARAMETERS	DESCRIPTION
SystemCapabilityType.DISPLAYS	Specifies display related information. The primary display will be the first element within the array. Windows within that display are different places that the app could be displayed (such as the main app window and various widget windows).
SystemCapabilityType.HMI_ZONE	Specifies HMI Zones in the vehicle. There may be a HMI available for back seat passengers as well as front seat passengers.
SystemCapabilityType.SPEECH	Contains information about TTS capabilities on the SDL platform. Platforms may support text, SAPI phonemes, LH PLUS phonemes, pre-recorded speech, and silence.
prerecordedSpeechCapabilities	Currently only available in the SDL_iOS library
SystemCapabilityType.VOICE_RECOGNITION	The voice-recognition capabilities of the connected SDL platform. The platform may be able to recognize spoken text in the current language.
SystemCapabilityType.AUDIO_PASSTHROUGH	Describes the sampling rate, bits per sample, and audio types available.
SystemCapabilityType.PCM_STREAMING	Describes different audio type configurations for the audio PCM stream service, e.g. {8kHz,8-bit,PCM}.
SystemCapabilityType.HMI	Returns whether or not the app can support built-in navigation and phone calls.

PARAMETERS	DESCRIPTION
SystemCapabilityType.APP_SERVICES	Describes the capabilities of app services including what service types are supported and the current state of services.
SystemCapabilityType.NAVIGATION	Describes the built-in vehicle navigation system's APIs.
SystemCapabilityType.PHONE_CALL	Describes the built-in phone calling capabilities of the IVI system.
SystemCapabilityType.VIDEO_STREAMING	Describes the abilities of the head unit to video stream projection applications.
SystemCapabilityType.REMOTE_CONTROL	Describes the abilities of an app to control built-in aspects of the IVI system.

Deprecated Properties

The following properties are deprecated on SDL Android 4.10 because as of RPC v6.0 they are deprecated. However, these properties will still be filled with information. When connected on RPC <6.0, the information will be exactly the same as what is returned in the `RegisterAppInterfaceResponse` and `SetDisplayLayoutResponse`. However, if connected on RPC >6.0, the information will be converted from the newer-style display information, which means that some information will not be available.

PARAMETERS	DESCRIPTION
SystemCapabilityType.DISPLAY	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.
SystemCapabilityType.BUTTON	A list of available buttons and whether the buttons support long, short and up-down presses.
SystemCapabilityType.SOFTBUTTON	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.
SystemCapabilityType.PRESET_BANK	If returned, the platform supports custom on-screen presets.

Image Specifics

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

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

```
ImageField field =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().get

ImageResolution resolution = field.getImageResolution();
```

EXAMPLE IMAGE SIZES

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

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

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

Querying for System Capabilities

Most head units provide features that your app can use: making and receiving phone calls, an embedded navigation system, video and audio streaming, as well as supporting app services. To find out if the head unit supports a feature as well as more information about the feature, use the `SystemCapabilityManager` to query the head unit for the desired capability. If a capability is unavailable, the query will return `null`.

```
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.A
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        AppServicesCapabilities servicesCapabilities = (AppServicesCapabilities)
capability;
    }

    @Override
    public void onError(String info) {
        <# Handle Error #>
    }
});
```

Subscribing to System Capabilities

In addition getting the current system capabilities, it is also possible to subscribe for updates when the head unit capabilities change. Since this information must be queried from Core you must implement the `OnSystemCapabilityListener`. This feature is only available RPC v5.1 or greater connections (except for DISPLAYS, which is backward compatible to RPC v1.0).

SUBSCRIBE TO A CAPABILITY

```
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(Syste
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        AppServicesCapabilities servicesCapabilities = (AppServicesCapabilities)
capability;
    }

    @Override
    public void onError(String info) {
        <# Handle Error #>
    }
});

GetSystemCapability getSystemCapability = new GetSystemCapability();
getSystemCapability.setSystemCapabilityType(SystemCapabilityType.APP_SERVI(

getSystemCapability.setSubscribe(true);
sdlManager.sendRPC(getSystemCapability);
```

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.getSystemCapabilityManager().getDefaultMainWindowCapability().getTemplatesAvailable()` .

Change the Template

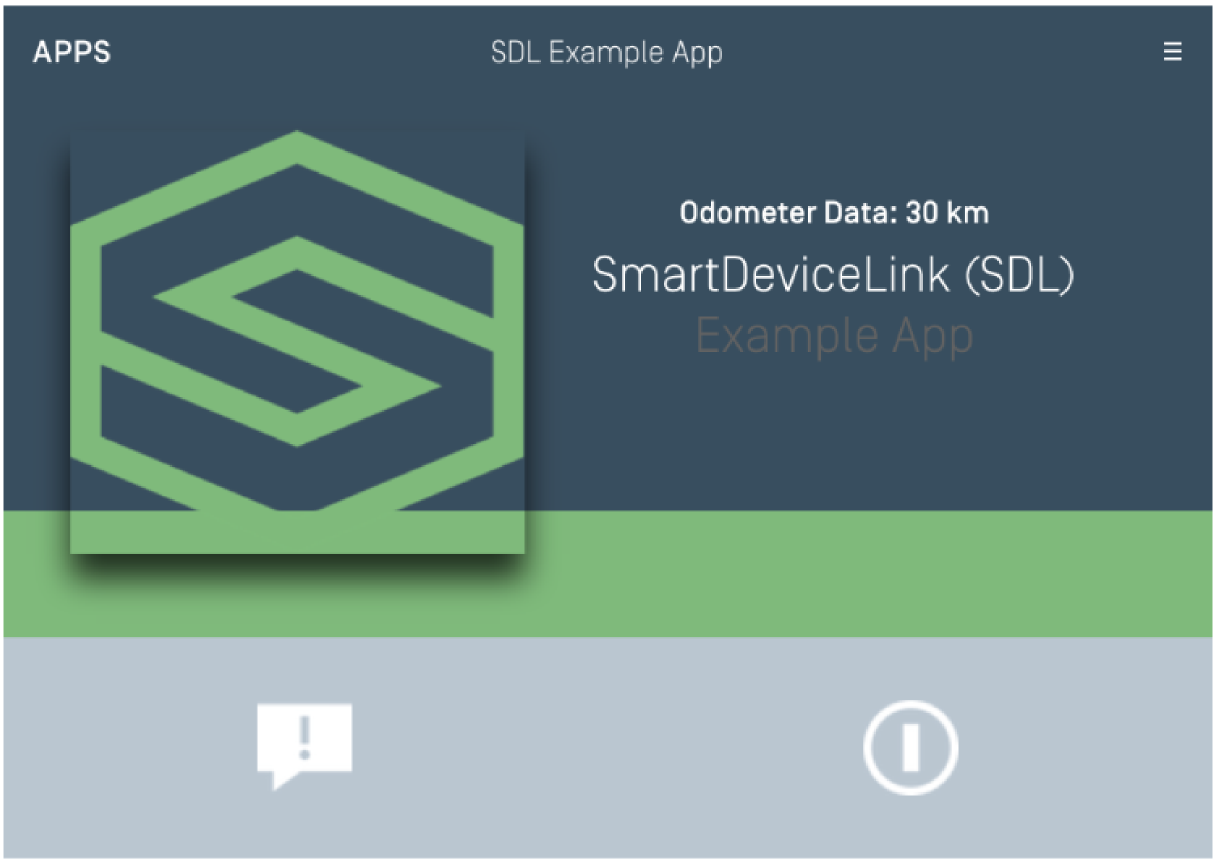
To change a template at any time, send a `SetDisplayLayout` RPC to Core.

```
SetDisplayLayout setDisplayLayoutRequest = new SetDisplayLayout();
setDisplayLayoutRequest.setDisplayLayout(PredefinedLayout.GRAPHIC_WITH_TE

setDisplayLayoutRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (((SetDisplayLayoutResponse) response).getSuccess()) {
            Log.i("SdlService", "Display layout set successfully.");
            // Proceed with more user interface RPCs
        } else {
            Log.i("SdlService", "Display layout request rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(setDisplayLayoutRequest);
```

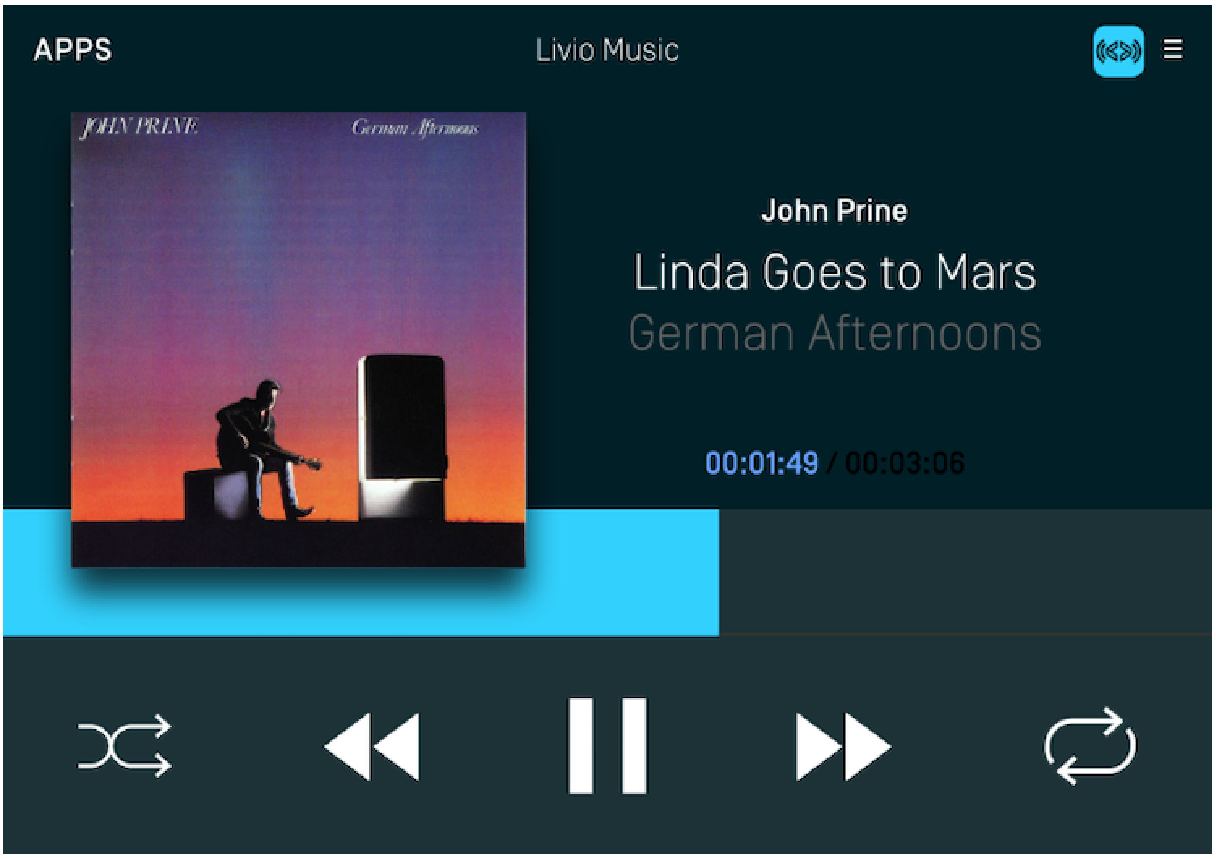
Available Templates

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

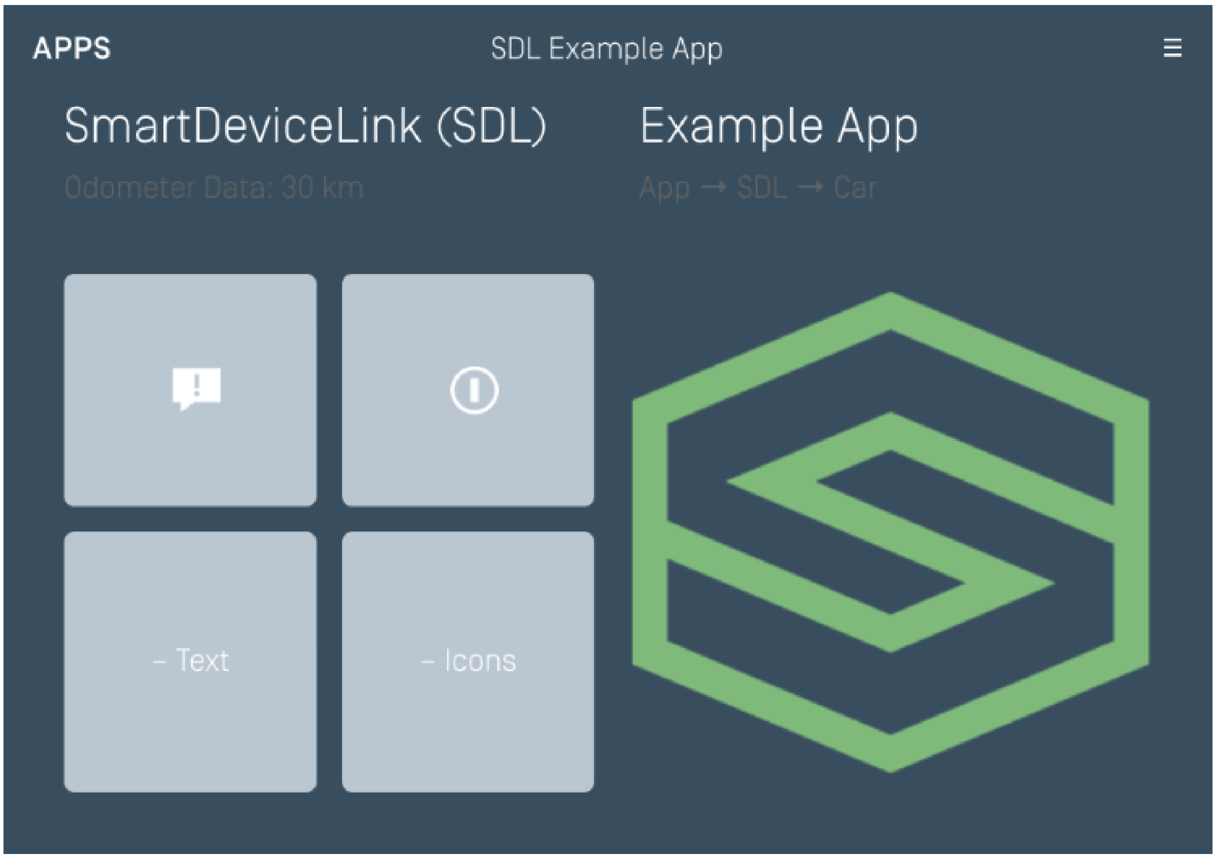
MEDIA



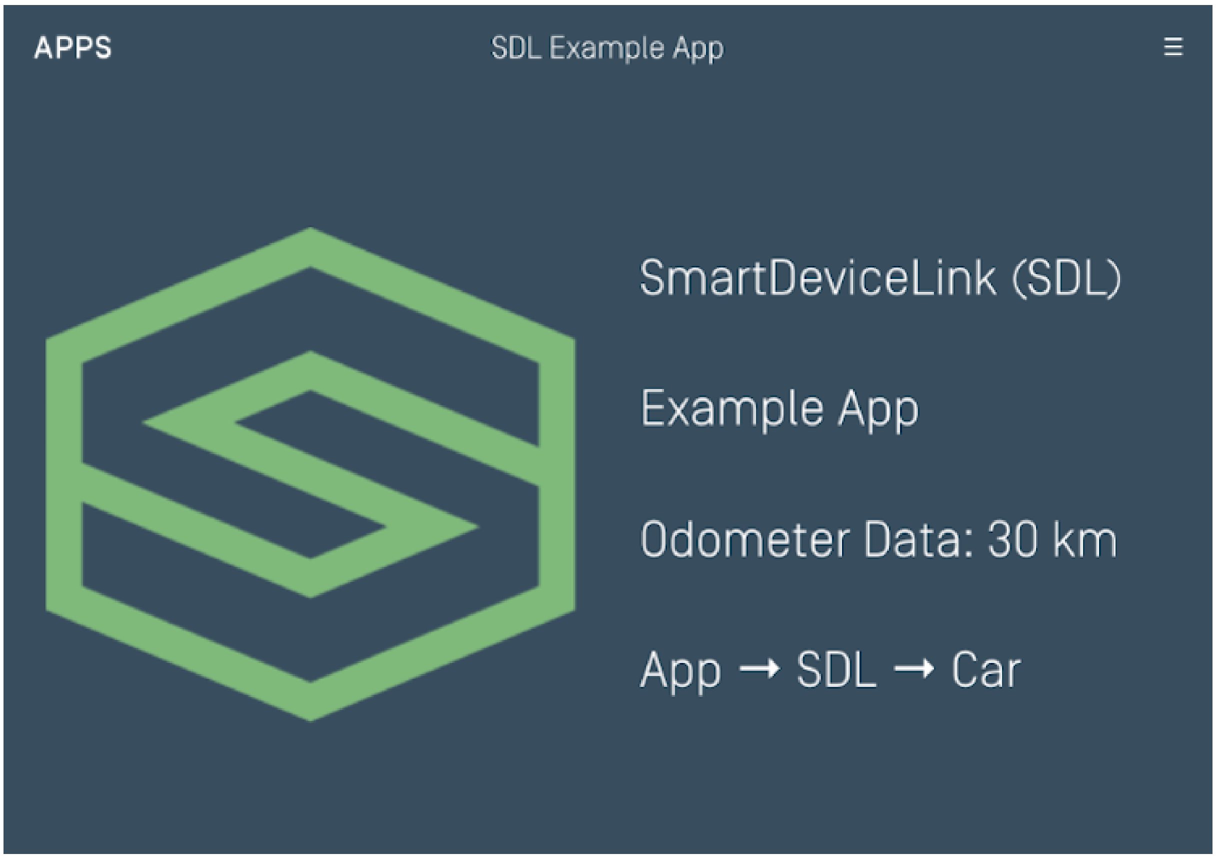
MEDIA (WITH A PROGRESS BAR)


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

NON-MEDIA



GRAPHIC WITH TEXT



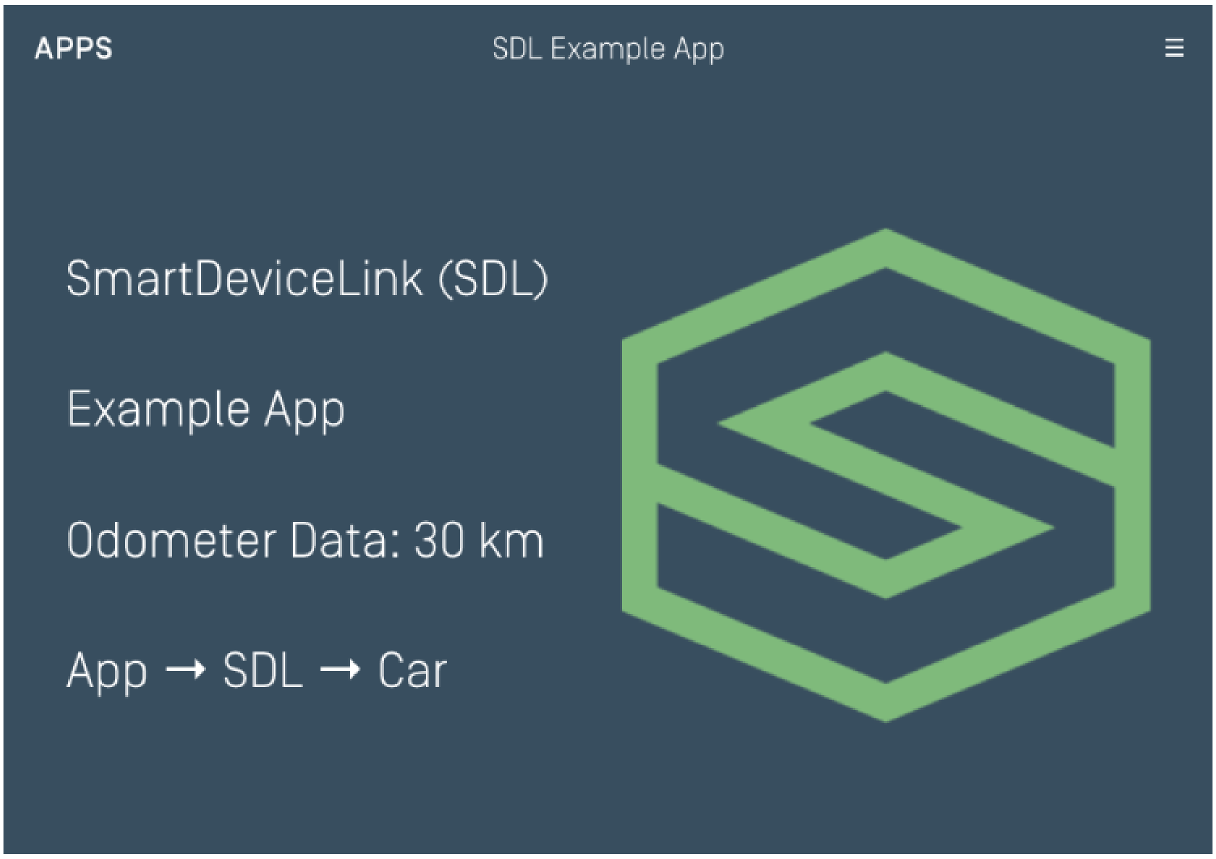
SmartDeviceLink (SDL)

Example App

Odometer Data: 30 km

App → SDL → Car

TEXT WITH GRAPHIC


APPS

SDL Example App

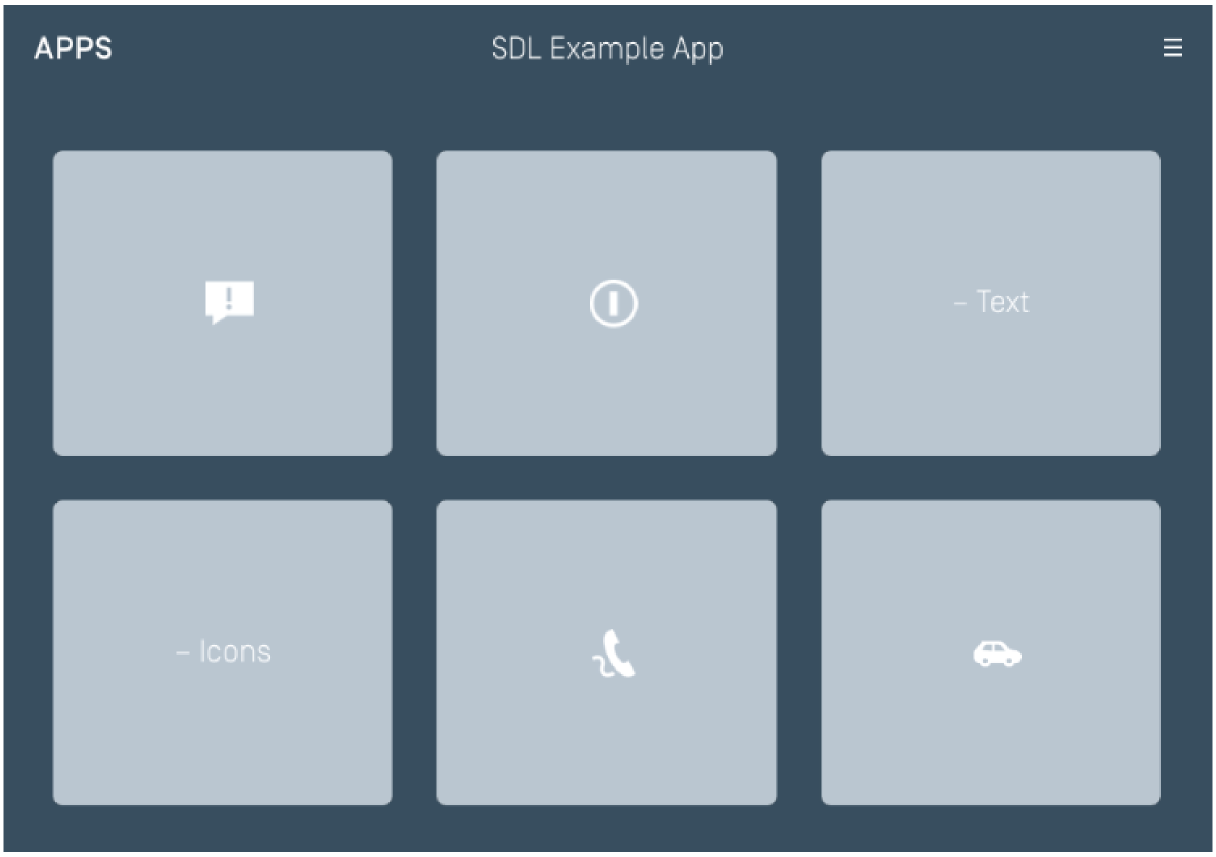
SmartDeviceLink (SDL)

Example App

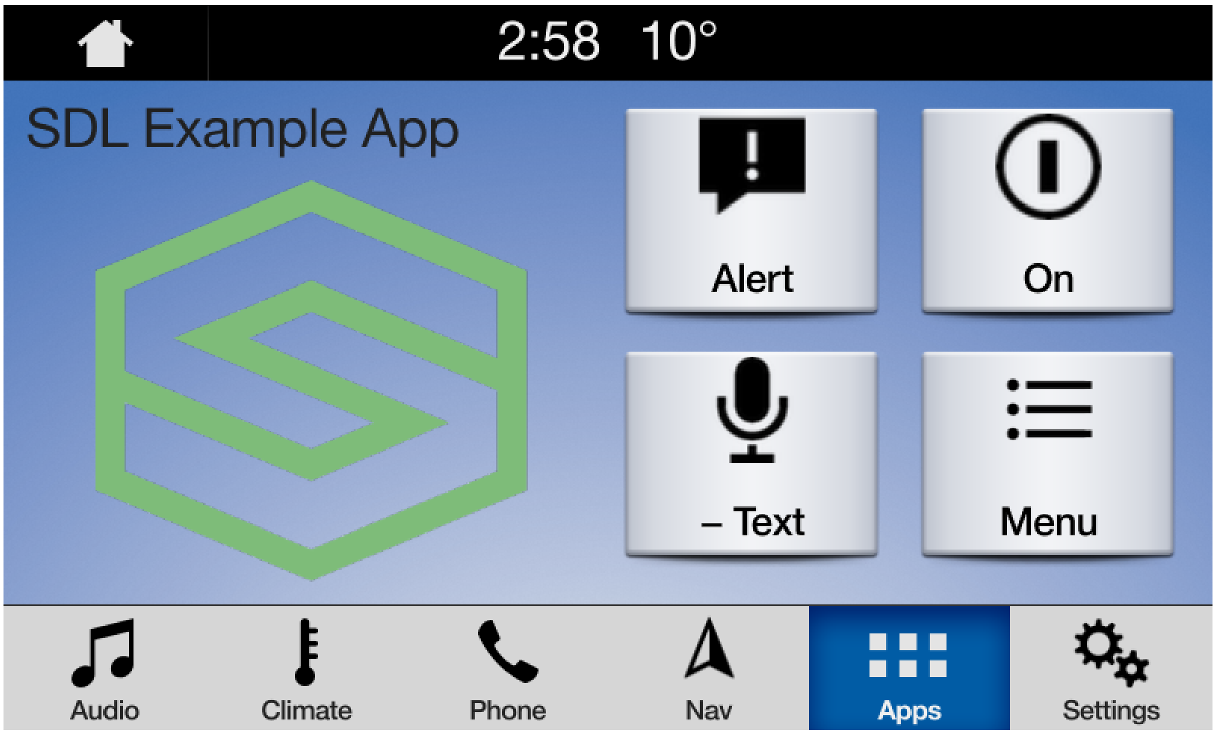
Odometer Data: 30 km

App → SDL → Car

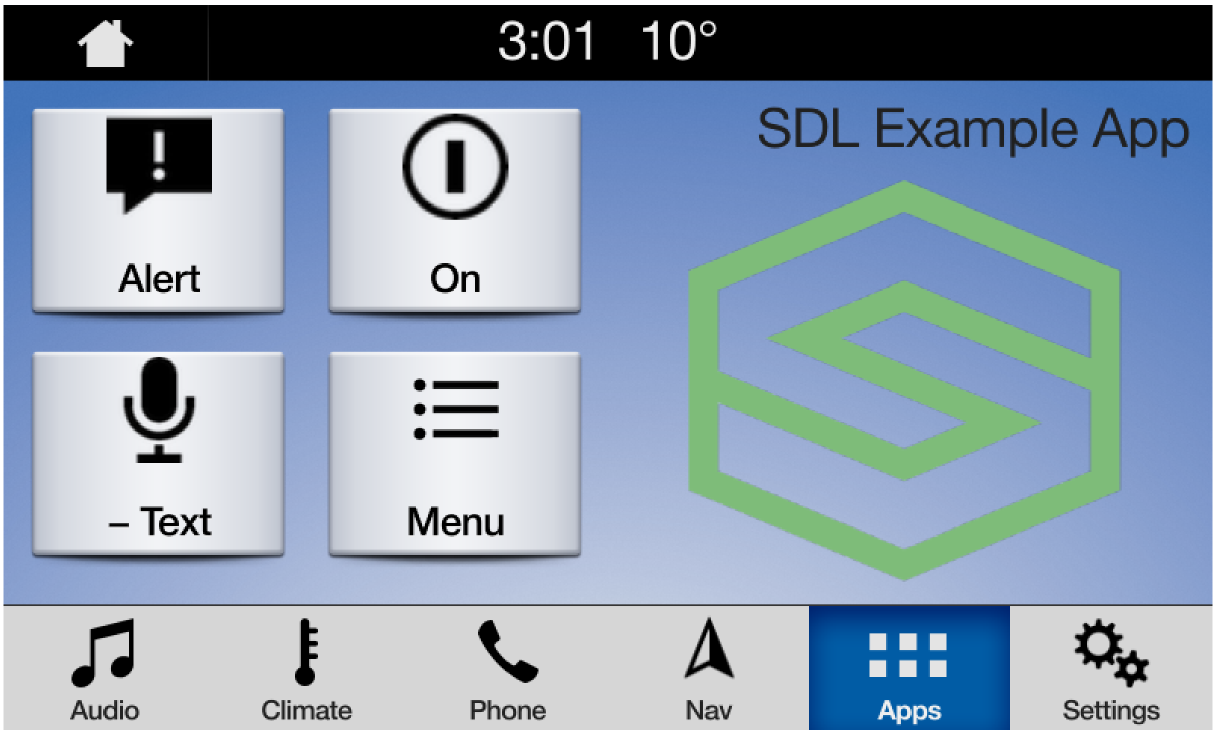
TILES ONLY



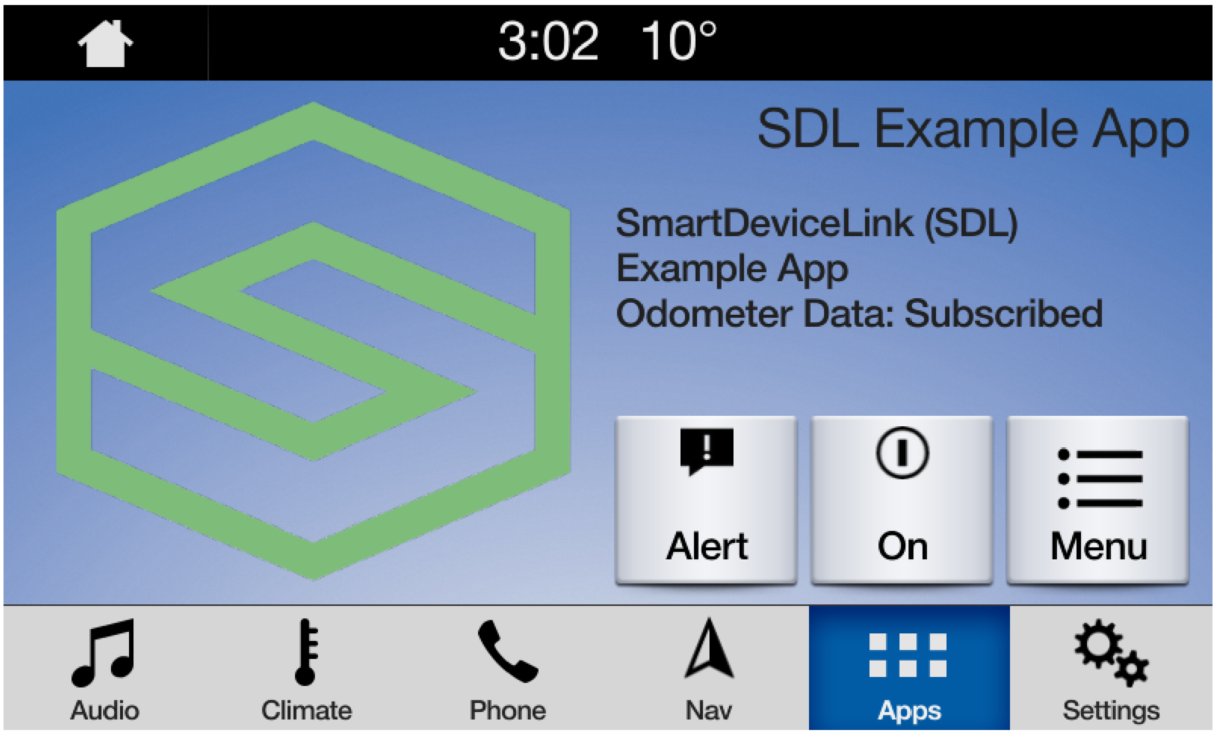
GRAPHIC WITH TILES



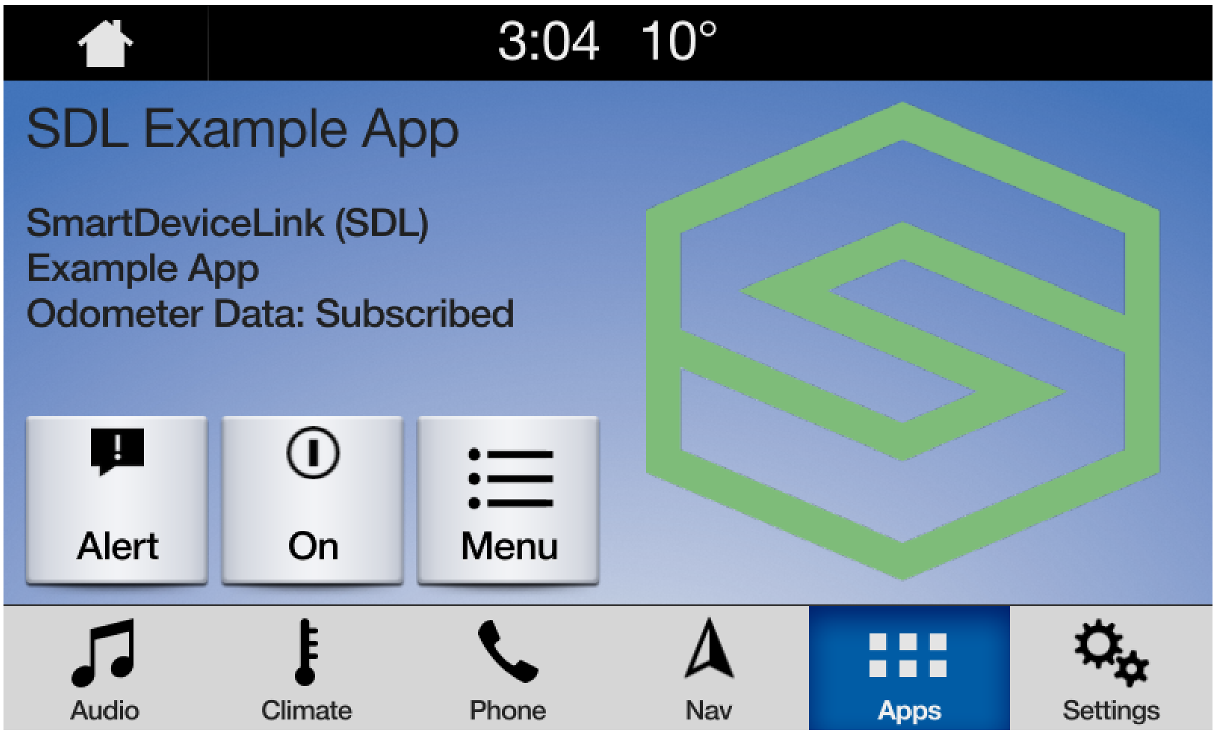
TILES WITH GRAPHIC



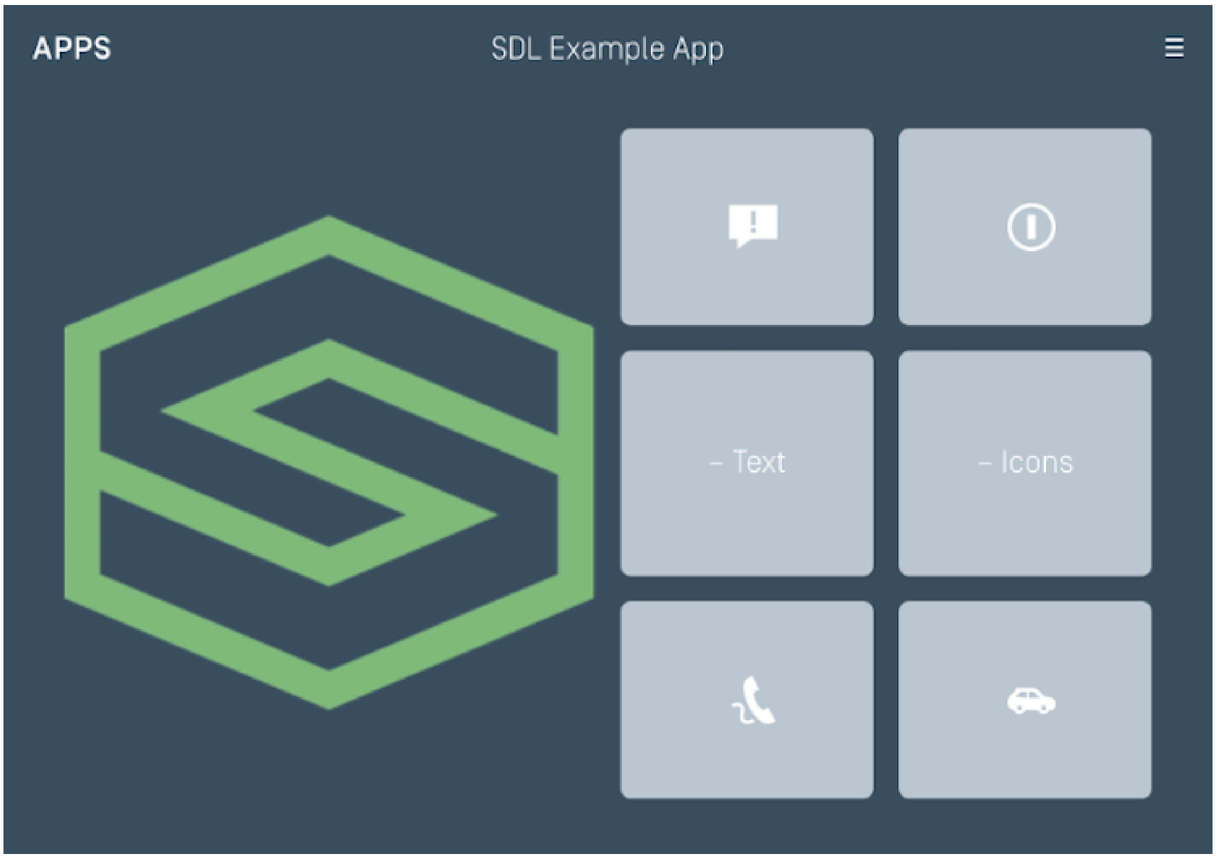
GRAPHIC WITH TEXT AND SOFT BUTTONS



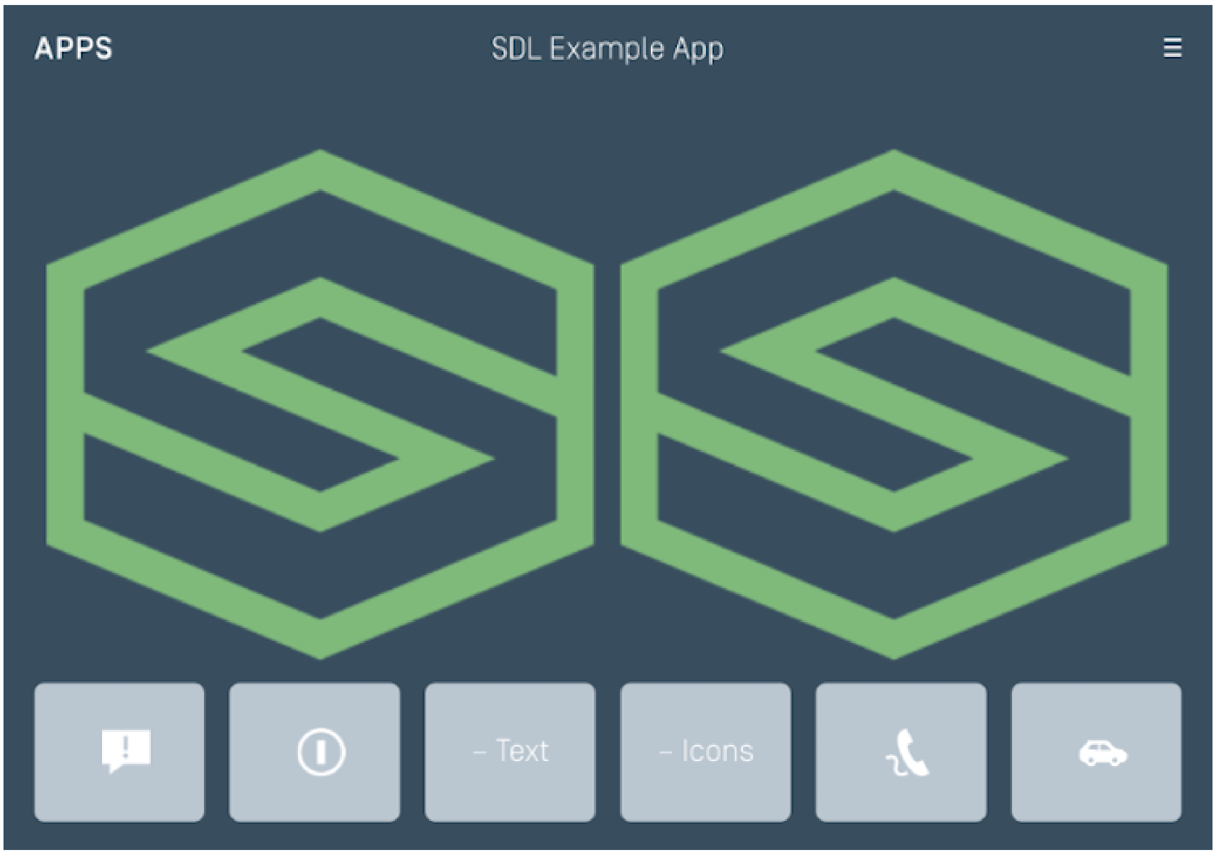
TEXT AND SOFT BUTTONS WITH GRAPHIC



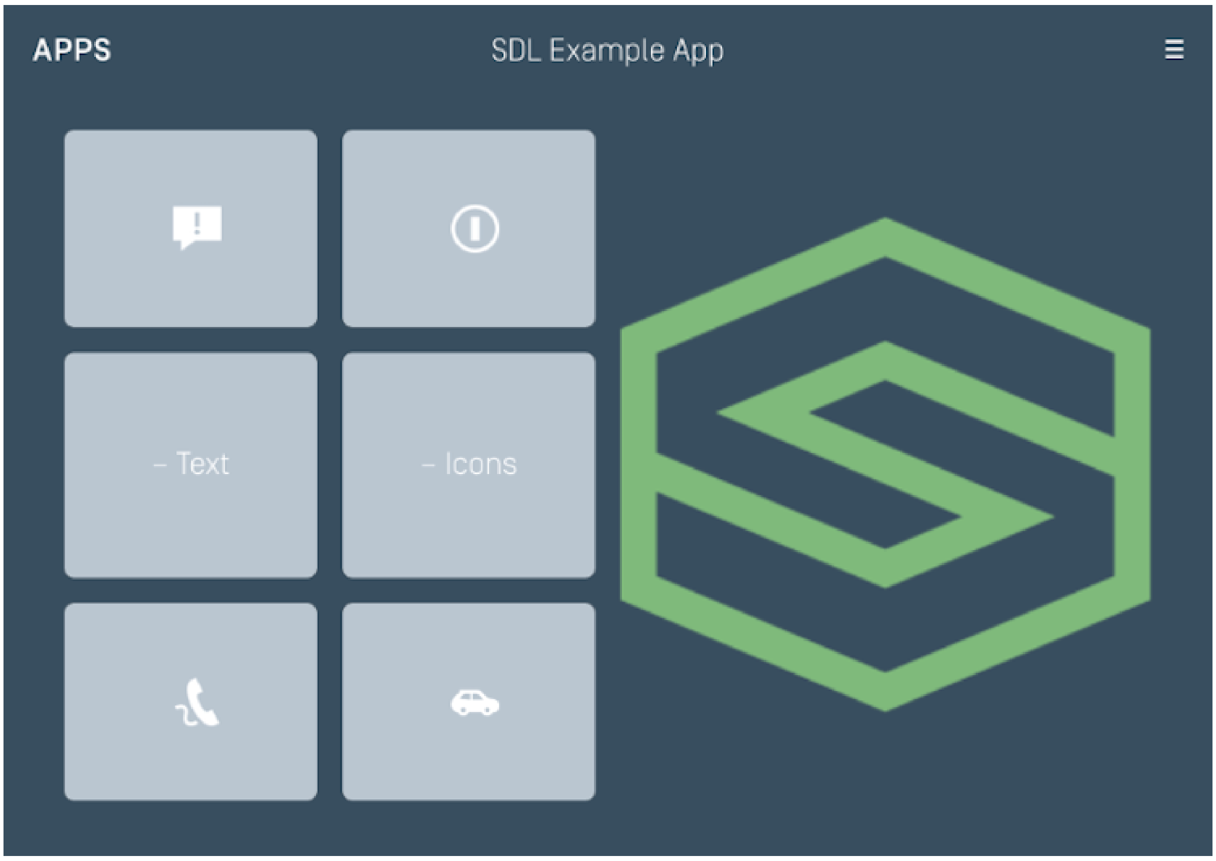
GRAPHIC WITH TEXT BUTTONS



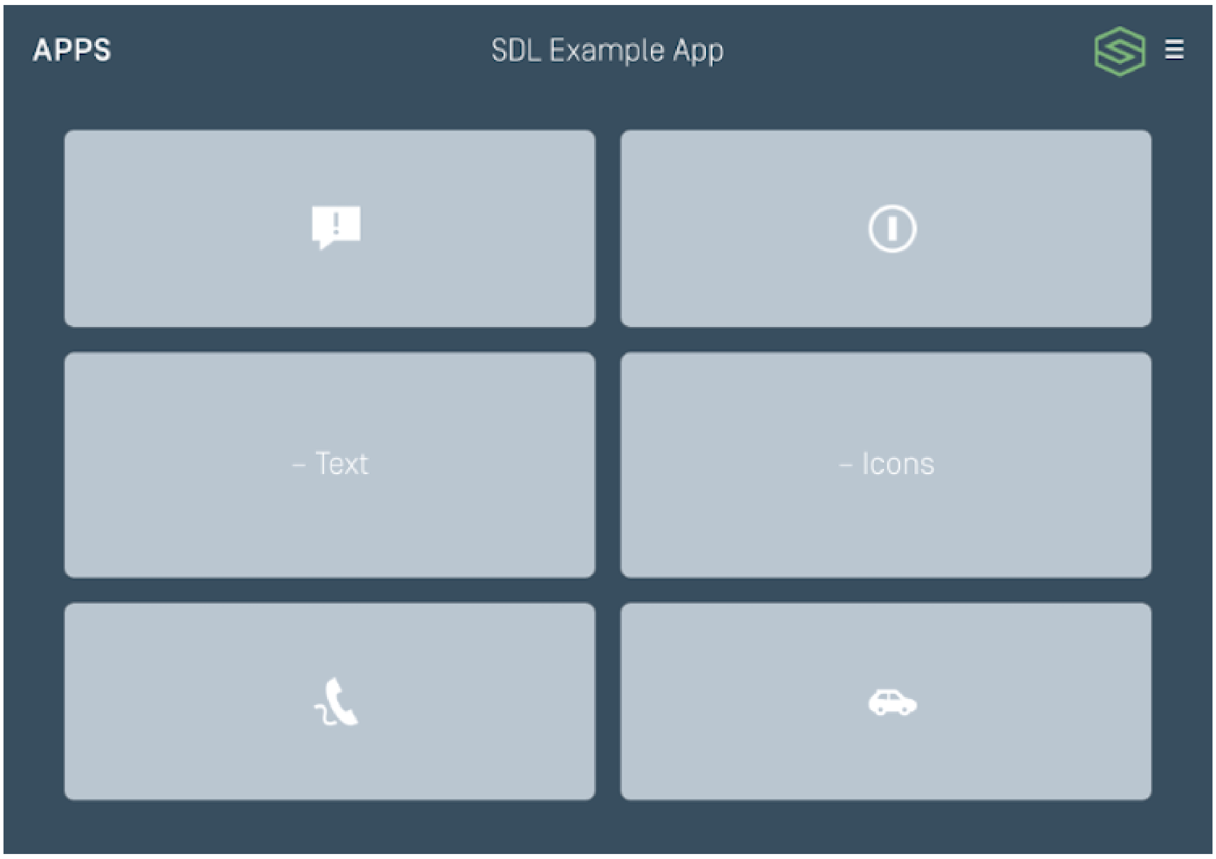
DOUBLE GRAPHIC WITH SOFT BUTTONS



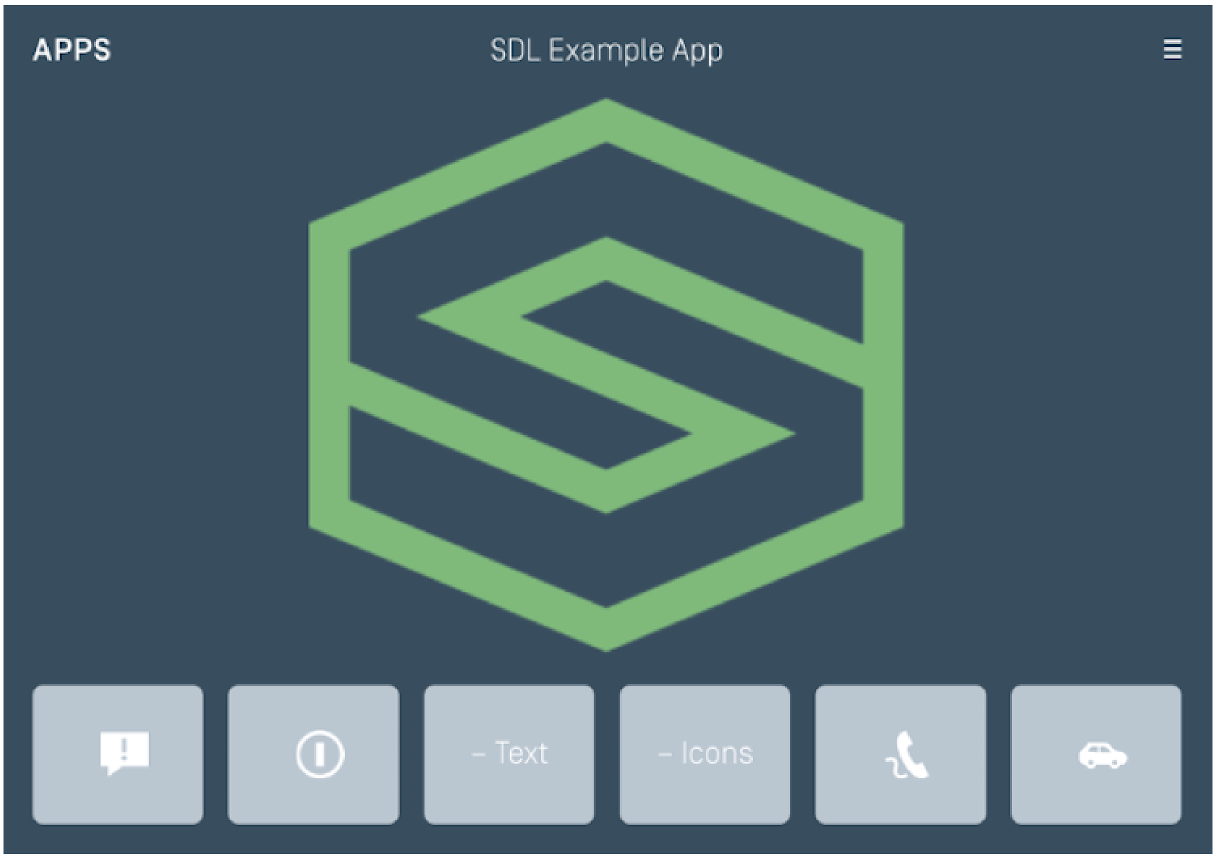
TEXT BUTTONS WITH GRAPHIC



TEXT BUTTONS ONLY



LARGE GRAPHIC WITH SOFT BUTTONS



LARGE GRAPHIC ONLY

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 `ScreenManager` 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 `beginTransaction()` and `commit()` 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
title	The title of the displayed template
mediaTrackTextField	The text displayed in the in the track field. This field is only valid for media applications
primaryGraphic	The primary image in a template that supports images
secondaryGraphic	The second image in a template that supports multiple images
textAlignment	The text justification for the text fields. The text alignment can be left, center, or right
softButtonObjects	An array of buttons. Each template supports a different number of soft buttons
textField1Type	The type of data provided in `textField1`
textField2Type	The type of data provided in `textField2`

SDLSCREENMANAGER PARAMETER NAME	DESCRIPTION
textField3Type	The type of data provided in `textField3`
textField4Type	The type of data provided in `textField4`

```
sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setTextField1("Hello, this is MainField1.");
sdlManager.getScreenManager().setTextField2("Hello, this is MainField2.");
sdlManager.getScreenManager().setTextField3("Hello, this is MainField3.");
sdlManager.getScreenManager().setTextField4("Hello, this is MainField4.");
sdlManager.getScreenManager().setTitle("<#Title#>");
sdlManager.getScreenManager().commit(new CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        Log.i(TAG, "ScreenManager update complete: " + success);
    }
});
```

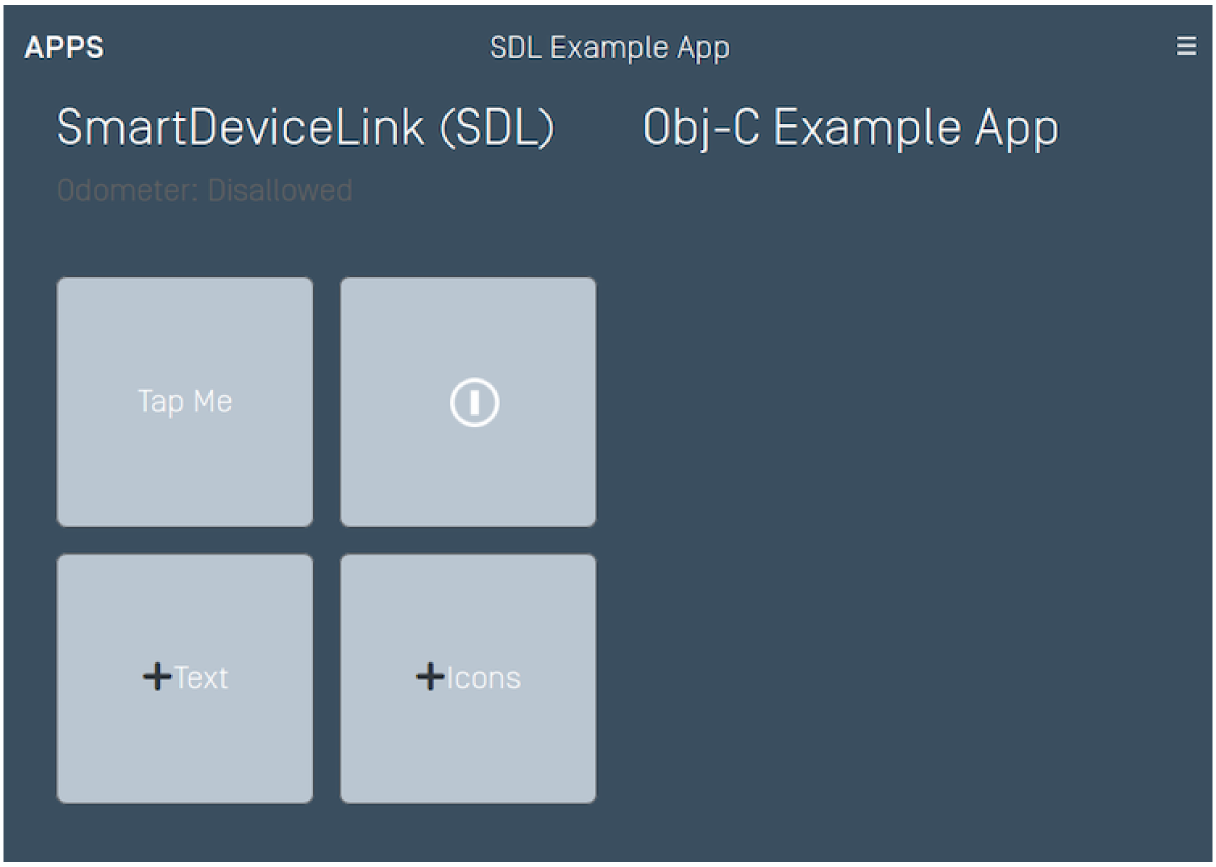
Removing Text and Images

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 `null`.

```
sdlManager.getScreenManager().setTextField1(null);
sdlManager.getScreenManager().setTextField2(null);
sdlManager.getScreenManager().setPrimaryGraphic(null);
sdlManager.getScreenManager().setTitle(null);
```

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)

```
SoftButtonState textState = new SoftButtonState("<#State Name#>", "<#Button
Label Text#>", null);
SoftButtonObject softButtonObject = new SoftButtonObject("softButtonObject",
Collections.singletonList(textState), textState.getName(), new
SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject, OnButtonPress
onButtonPress) {
    }

    @Override
    public void onEvent(SoftButtonObject softButtonObject, OnButtonEvent
onButtonEvent) {

    }
});

sdlManager.getScreenManager().setSoftButtonObjects(Collections.singletonList(
```

SOFT BUTTON (IMAGE ONLY)

To see if soft buttons support images you should check the `getGraphicSupported()` method on `SdlManager` s.

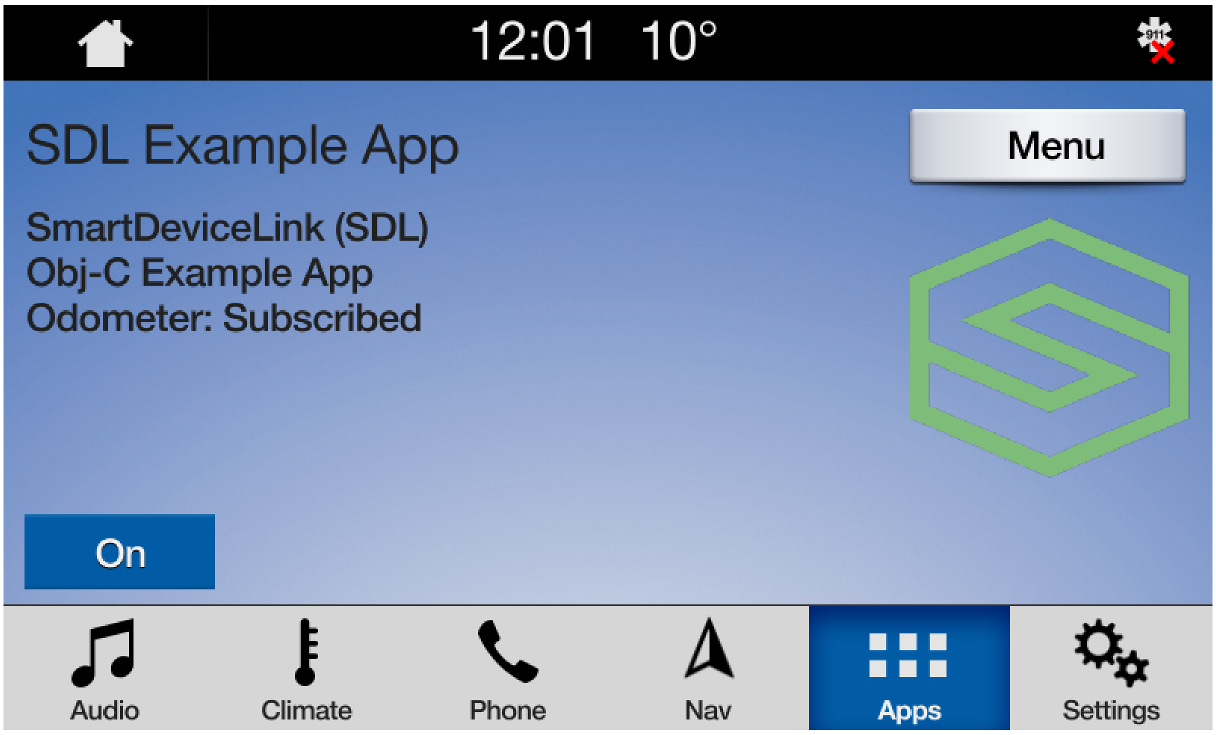
```
Object softButtonCapabilities =
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.S

List<SoftButtonCapabilities> softButtonCapabilitiesList =
SystemCapabilityManager.convertToList(softButtonCapabilities,
SoftButtonCapabilities.class);
boolean imageSupported = false;
if (softButtonCapabilities != null && !softButtonCapabilitiesList.isEmpty() &&
softButtonCapabilitiesList.get(0).getImageSupported()){
    imageSupported = true;
}


if (imageSupported) {
    SoftButtonState state = new SoftButtonState("<#State Name#>", null,
imageArtwork);
    SoftButtonObject softButtonObject = new SoftButtonObject("softButtonObject",
Collections.singletonList(state), state.getName(), new
SoftButtonObject.OnEventListener() {
        @Override
        public void onPress(SoftButtonObject softButtonObject, OnButtonPress
onButtonPress) {
        }

        @Override
        public void onEvent(SoftButtonObject softButtonObject, OnButtonEvent
onButtonEvent) {

        }
    });


sdlManager.getScreenManager().setSoftButtonObjects(Collections.singletonList(

}
```

SOFT BUTTON (IMAGE AND TEXT)

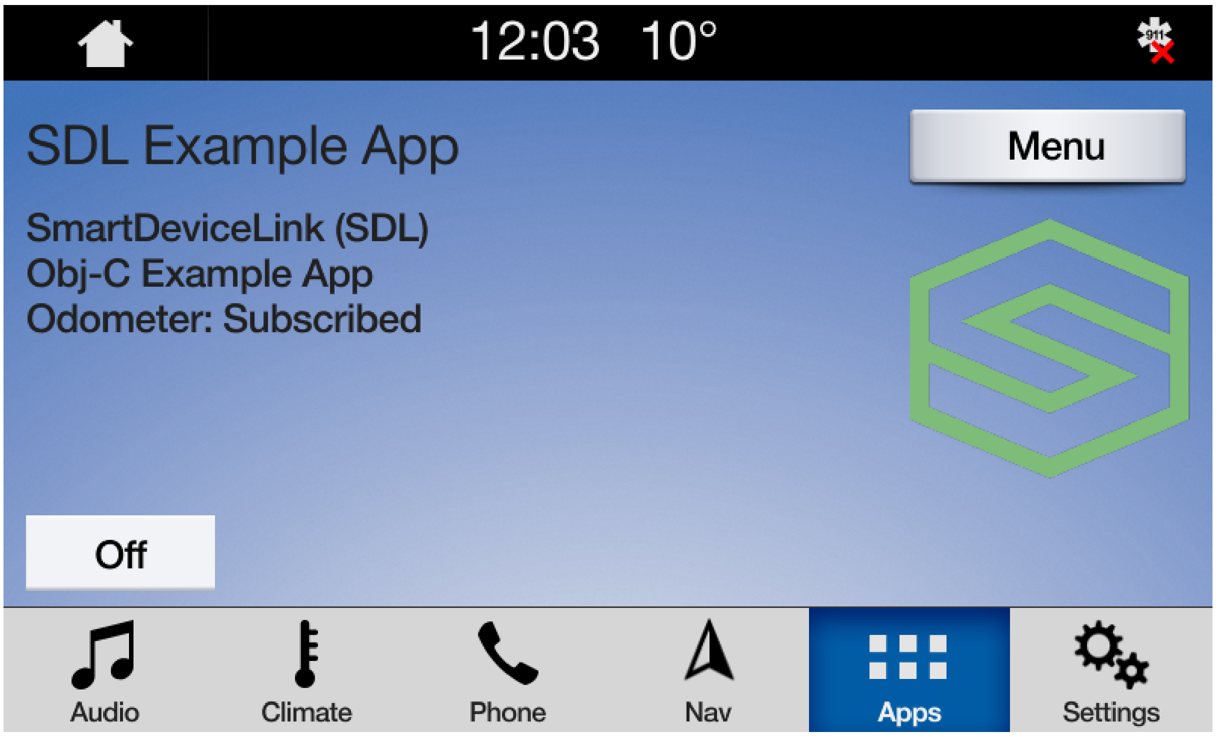
```
SoftButtonState state = new SoftButtonState("<#State Name#>", "<#Button Label
Text#>", imageArtwork);
SoftButtonObject softButtonObject = new SoftButtonObject("softButtonObject",
Collections.singletonList(state), state.getName(), new
SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject, OnButtonPress
onButtonPress) {
    }

    @Override
    public void onEvent(SoftButtonObject softButtonObject, OnButtonEvent
onButtonEvent) {

    }
});

sdlManager.getScreenManager().setSoftButtonObjects(Collections.singletonList(
```

Highlighting the Soft Button

HIGHLIGHT ON



HIGHLIGHT OFF



```
SoftButtonState softButtonState1 = new SoftButtonState("Soft Button State
Name", "On", image1Artwork);
softButtonState1.setHighlighted(true);
SoftButtonState softButtonState2 = new SoftButtonState("Soft Button State
Name 2", "Off", image2Artwork);
softButtonState2.setHighlighted(false);
SoftButtonObject softButtonObject = new SoftButtonObject("softButtonObject",
Arrays.asList(softButtonState1, softButtonState2), softButtonState1.getName(),
new SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject, OnButtonPress
onButtonPress) {
        softButtonObject.transitionToNextState();
    }

    @Override
    public void onEvent(SoftButtonObject softButtonObject, OnButtonEvent
onButtonEvent) {

    }
});
```

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.

```
SoftButtonState state1 = new SoftButtonState("<#State1 Name#>", "<#Button1
Label Text#>", image1Artwork);
SoftButtonState state2 = new SoftButtonState("<#State2 Name#>", "<#Button2
Label Text#>", image2Artwork);

SoftButtonObject softButtonObject = new SoftButtonObject("softButtonObject",
Arrays.asList(state1, state2), state1.getName(), new
SoftButtonObject.OnEventListener() {
    @Override
    public void onPress(SoftButtonObject softButtonObject, OnButtonPress
onButtonPress) {

    }

    @Override
    public void onEvent(SoftButtonObject softButtonObject, OnButtonEvent
onButtonEvent) {

    }
});

sdlManager.getScreenManager().setSoftButtonObjects(Collections.singletonList(



// Transition to a new state
SoftButtonObject retrievedSoftButtonObject =
sdlManager.getScreenManager().getSoftButtonObjectByName("softButtonObject")

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.

```
sdlManager.getScreenManager().setSoftButtonObjects(Collections.EMPTY_LIST);
```

Templating Images

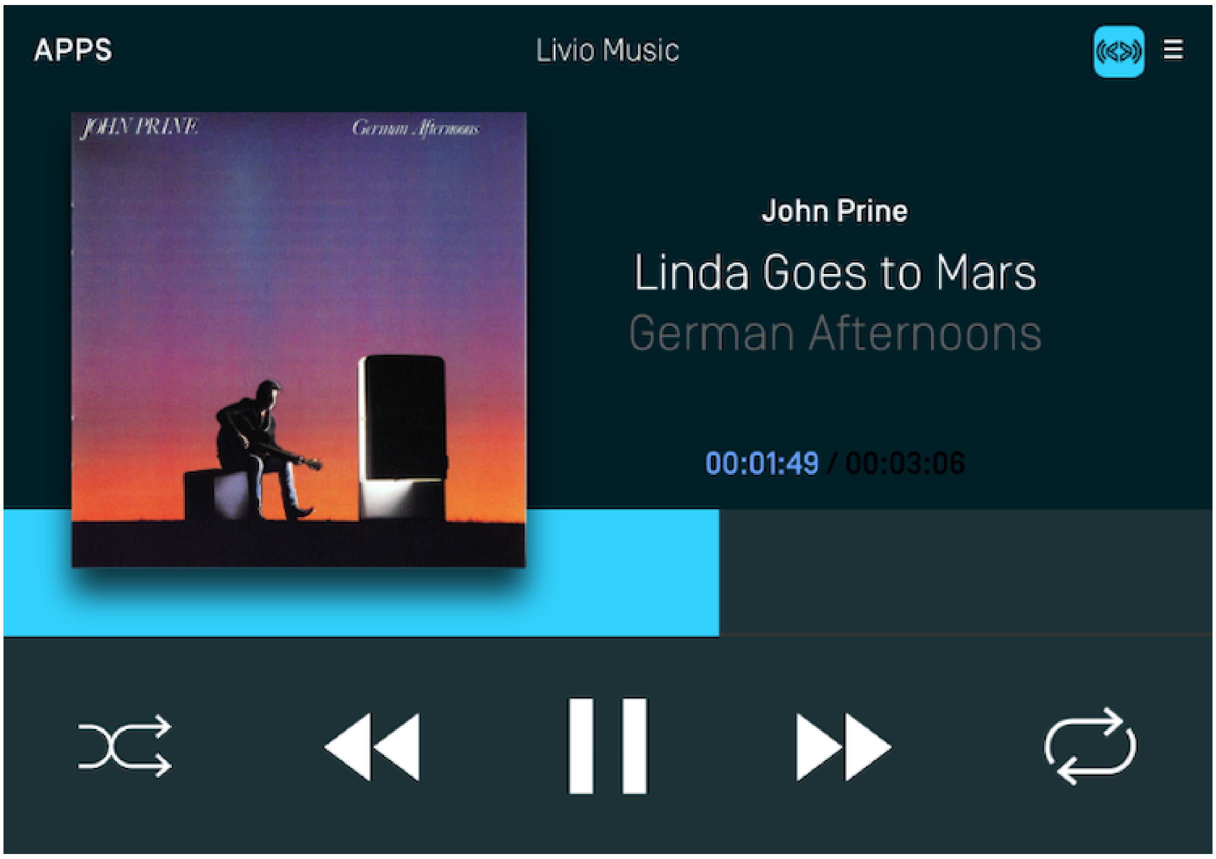
When connected to a remote system running SDL Core 5.0+, you may be able to use template images. Templated images are tinted by Core so the image is visible regardless of whether your user has set the head unit to day or night mode. For example, if a head unit is in night mode with a dark theme (see 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. 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.

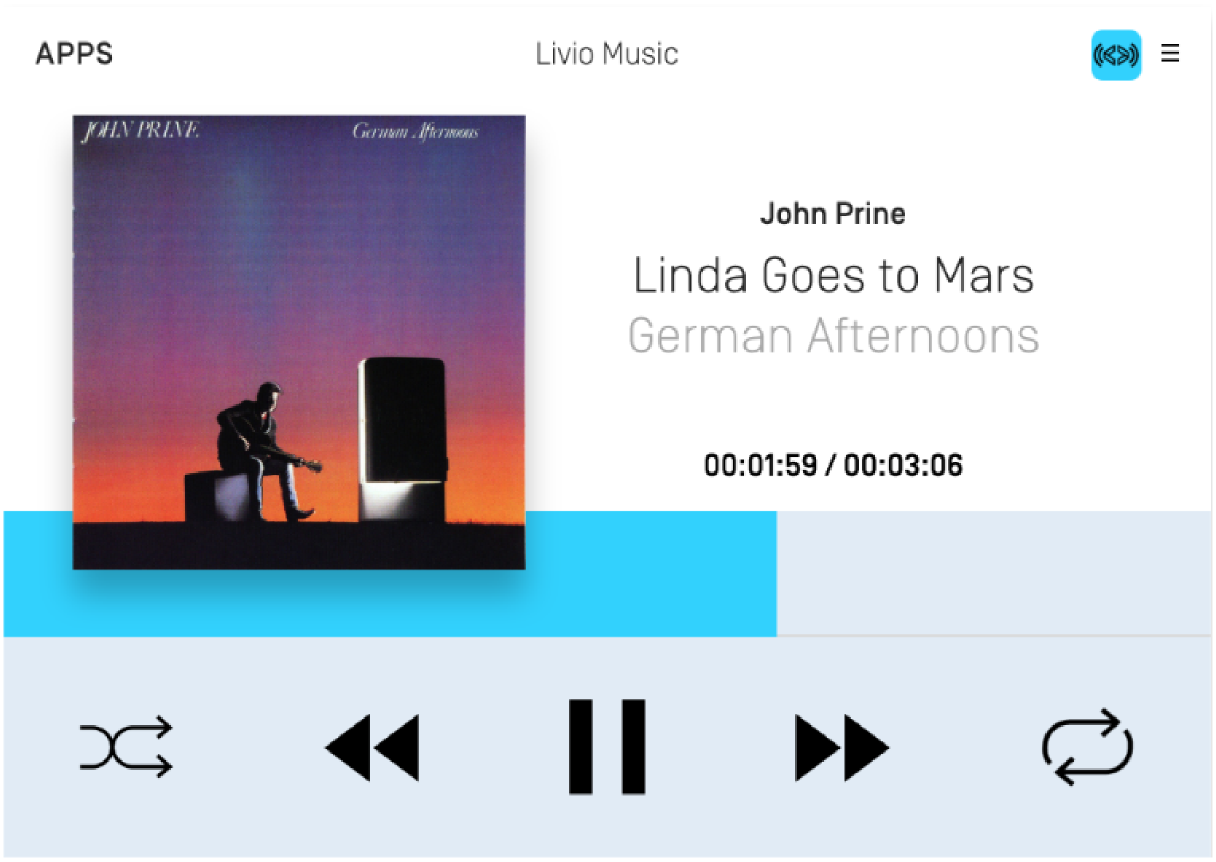
TEMPLATED IMAGES EXAMPLE

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



```
SdlArtwork image = new SdlArtwork("ArtworkName", FileType.GRAPHIC_PNG,
R.drawable.artworkName, true);
image.setTemplateImage(true);
```

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.

```
SdlArtwork sdlArtwork = new SdlArtwork(StaticIconName.ALBUM);
sdlManager.getScreenManager().setPrimaryGraphic(sdlArtwork);
```

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
Play / Pause	Media only
Ok	Media only
Seek left	Media only
Seek right	Media only
Tune up	Media only
Tune down	Media only
Preset 0-9	Any
Search	Any
Center Location	NavigationFullscreenMap only
Zoom In	NavigationFullscreenMap only
Zoom Out	NavigationFullscreenMap only
Pan Up	NavigationFullscreenMap only
Pan Up-Right	NavigationFullscreenMap only
Pan Right	NavigationFullscreenMap only
Pan Down-Right	NavigationFullscreenMap only
Pan Down	NavigationFullscreenMap only

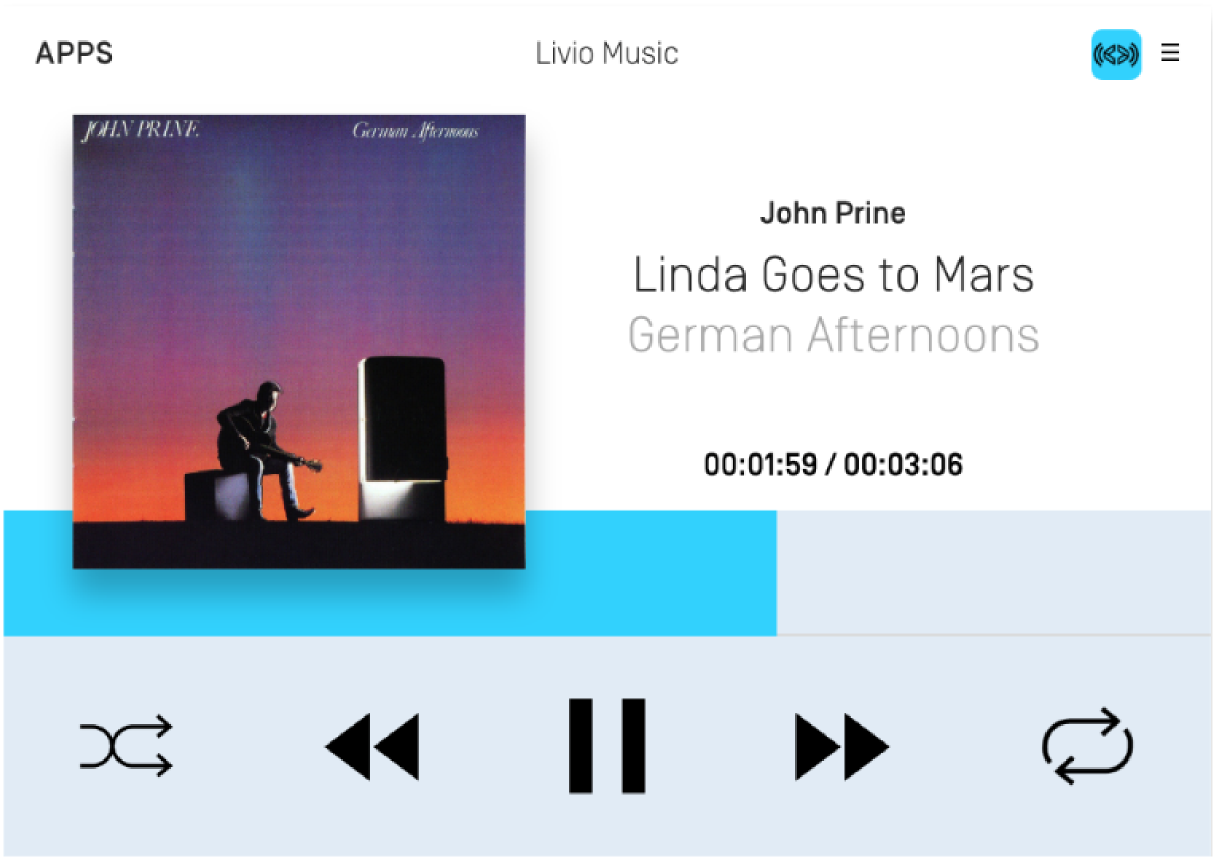
BUTTON	TEMPLATE
Pan Down-Left	NavigationFullscreenMap only
Pan Left	NavigationFullscreenMap only
Pan Up-Left	NavigationFullscreenMap only
Toggle Tilt	NavigationFullscreenMap only
Rotate Clockwise	NavigationFullscreenMap only
Rotate Counter-Clockwise	NavigationFullscreenMap only
Toggle Heading	NavigationFullscreenMap only

NOTE

`Media` is the default template for media apps and `NavigationFullscreenMap` is the default template for navigation apps.

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

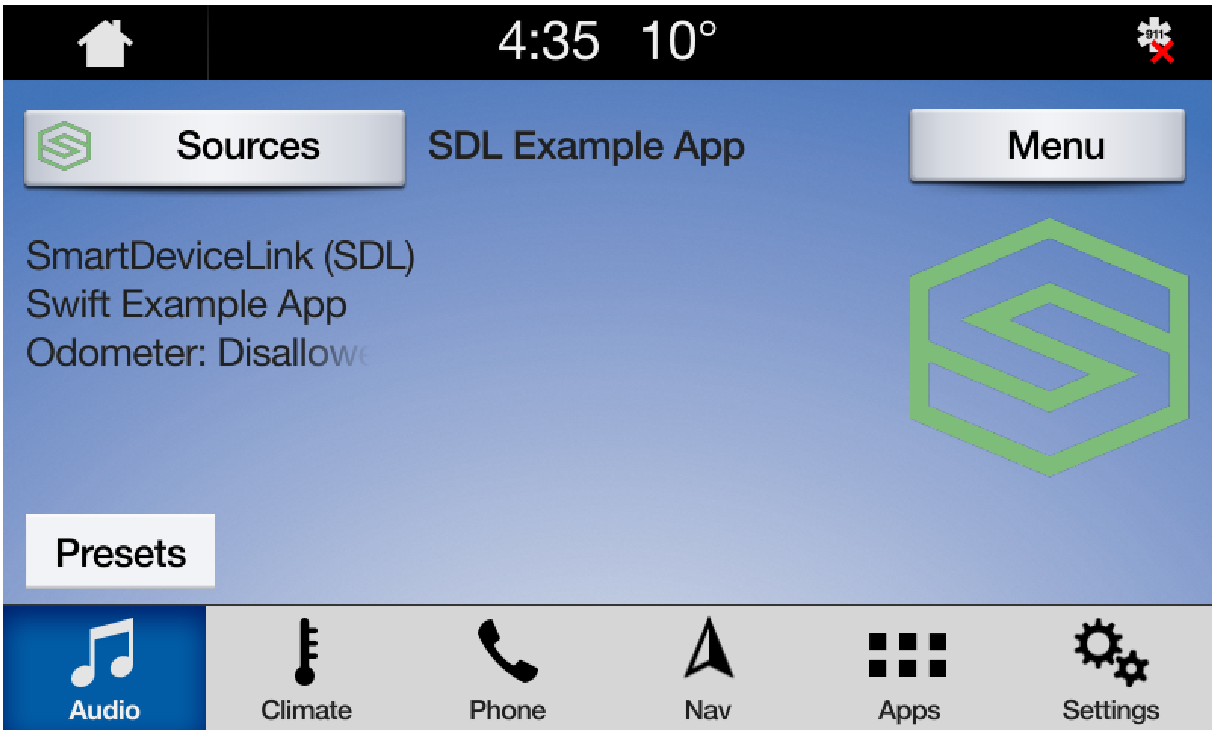
The play/pause, seek left, seek right, tune up, and tune down subscribe buttons can only be used in the `MEDIA` template. Depending on the manufacturer of the head unit, the subscribe button might also show up as a soft button in the media template. For example, the SYNC 3 HMI will add the ok, seek right, 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

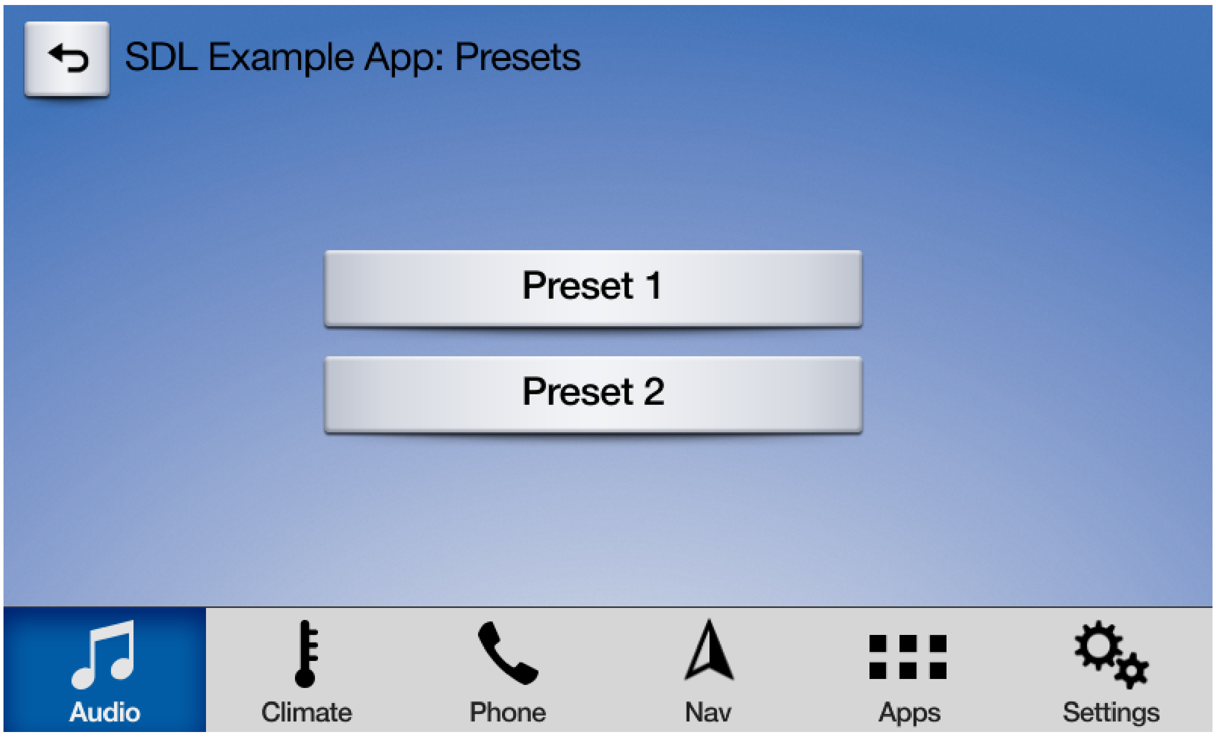
Before library v.4.7 and SDL Core v.5.0, `Ok` and `PlayPause` were combined into `Ok`. Subscribing to `Ok` will, in v.4.7, also subscribe you to `PlayPause`. This means that for the time being, *you should not simultaneously subscribe to `Ok` and `PlayPause`*. In a future major version, this will change. 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.

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_BUTTON_EVENT, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress) notification;
        switch (onButtonPressNotification.getButtonName()) {
            case OK:
                break;
            case PLAY_PAUSE:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
    }
});

sdlManager.addOnRPCNotificationListener(FunctionID.ON_BUTTON_PRESS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress) notification;
        switch (onButtonPressNotification.getButtonName()) {
            case OK:
                break;
            case PLAY_PAUSE:
                break;
            case SEEKLEFT:
                break;
            case SEEKRIGHT:
                break;
            case TUNEUP:
                break;
            case TUNEDOWN:
                break;
        }
    }
});

SubscribeButton subscribeButtonRequest = new SubscribeButton();
subscribeButtonRequest.setButtonName(ButtonName.OK);
sdlManager.sendRPC(subscribeButtonRequest);
```

Preset Buttons



SmartDeviceLink (SDL)
Swift Example App
Odometer: Disallowe



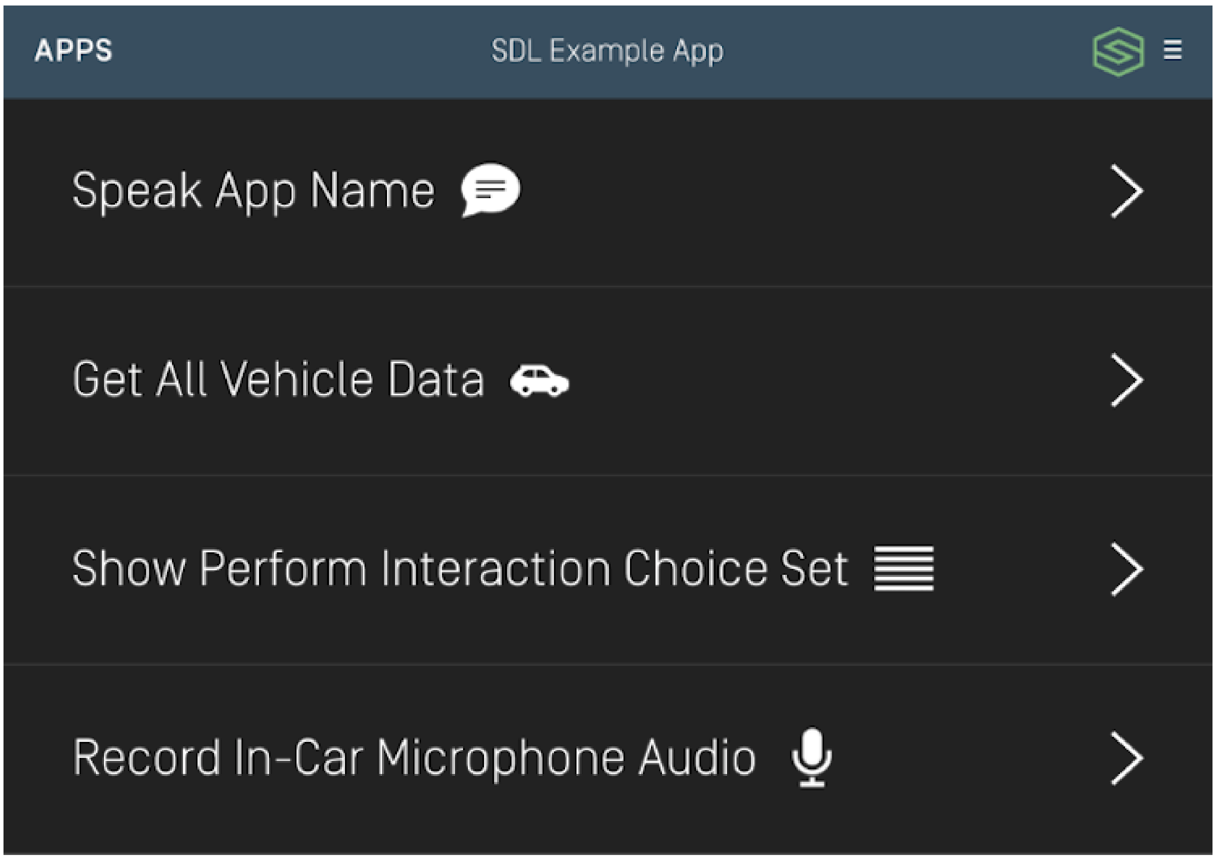
Preset buttons may not work in the same way as seen on the above screenshots on all head units. Some head units may have physical buttons on their console and these will trigger the subscribed button. You can check if an HMI supports subscribing to preset buttons, and how many, by calling the sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().getNumCuston
.

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_BUTTON_EVENT, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress) notification;
        switch (onButtonPressNotification.getButtonName()) {
            case PRESET_1:
                break;
            case PRESET_2:
                break;
        }
    }
});

sdlManager.addOnRPCNotificationListener(FunctionID.ON_BUTTON_PRESS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPressNotification = (OnButtonPress) notification;
        switch (onButtonPressNotification.getButtonName()) {
            case PRESET_1:
                break;
            case PRESET_2:
                break;
        }
    }
});

SubscribeButton preset1 = new SubscribeButton(ButtonName.PRESET_1);
SubscribeButton preset2 = new SubscribeButton(ButtonName.PRESET_2);
sdlManager.sendRPCs(Arrays.asList(preset1, preset2), null);
```

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


APPS
SDL Example App
Speak App Name
Get All Vehicle Data
Show Perform Interaction Choice Set
Record In-Car Microphone Audio

NOTE

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

Setting the Menu Layout (RPC v6.0+)

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

```
MenuConfiguration menuConfiguration = new
MenuConfiguration(<#mainMenuLayout>, <#submenuLayout>);
sdlManager.getScreenManager().setMenuConfiguration(menuConfiguration);
```

Adding Menu Items

The best way to create and update your menu is to the use the Screen Manager API. The screen manager contains two menu related properties: `menu`, and `voiceCommands`. Setting an array of `MenuCell`s into the `menu` property will automatically set and update your menu and submenus, while setting an array of `VoiceCommand`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.

```
// Create the menu cell
MenuCell cell = new MenuCell("Cell text", null, Collections.singletonList("cell
text"), new MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        // Menu item was selected, check the `triggerSource` to know if the user
used touch or voice to activate it
        // <#Handle the Cell's Selection#>
    }
});

sdlManager.getScreenManager().setMenu(Collections.singletonList(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. In RPC v6.0+ it is possible to set individual submenus to use different layouts such as tiles or lists.

```
// Create the inner menu cell
MenuCell innerCell = new MenuCell("inner menu cell", null,
Collections.singletonList("inner menu cell"), new MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        // 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
MenuCell cell = new MenuCell("cell", MenuLayout.LIST, null,
Collections.singletonList(innerCell));

sdlManager.getScreenManager().setMenu(Collections.singletonList(cell));
```

Menu Item Artwork

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

Deleting and Changing Menu Items

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

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

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

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

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

Presenting a Popup Menu

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

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

Creating Cells

An `ChoiceCell` is similar to a `RecyclerView` without the ability to configure your own UI. We provide several properties on the `ChoiceCell` 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.

```
ChoiceCell cell = new ChoiceCell("cell1 text", Collections.singletonList("cell1"),
null);
ChoiceCell fullCell = new ChoiceCell("cell2 text", "cell2 secondaryText", "cell2
tertiaryText", Collections.singletonList("cell2"), image1Artwork, image2Artwork);
```

Preloading Cells

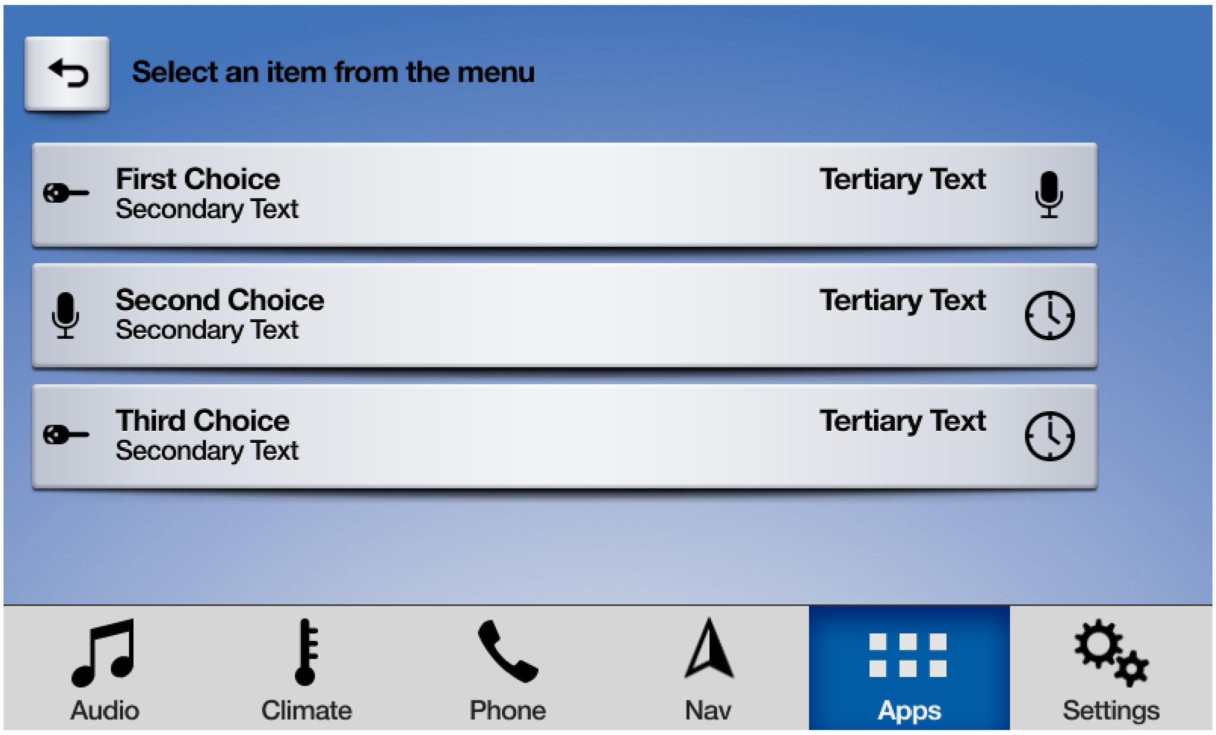
If you know the content you will show in the popup menu long before the menu is shown to the user, you can "preload" those cells in order to speed up the popup menu presentation at a later time. Once you preload a cell, you can reuse it in multiple popup menus without having to send the cell content to Core again.

```
sdlManager.getScreenManager().preloadChoices(Arrays.asList(cell, fullCell), new
CompletionListener() {
    @Override
    public void onComplete(boolean b) {
        // <#code#>
    }
});
```

Presenting a Menu

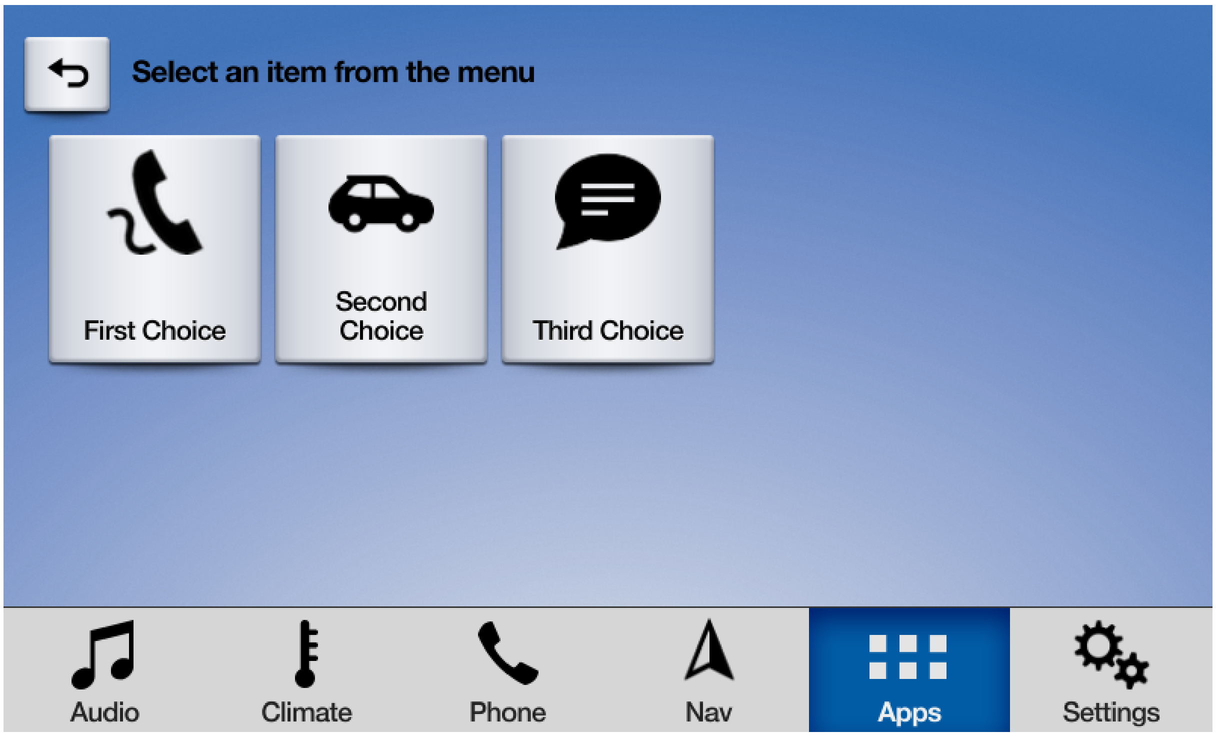
To show a popup menu to the user, you must present the menu. If some or all of the cells in the menu have not yet been preloaded, calling the `present` API will preload the cells and then present the menu once all the cells have been uploaded. Calling `present` without preloading the cells can take longer 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


Select an item from the menu

First Choice Secondary Text	Tertiary Text	
Second Choice Secondary Text	Tertiary Text	
Third Choice Secondary Text	Tertiary Text	

MENU - ICON


Select an item from the menu

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 `ChoiceCell`s into an `ChoiceSet`.

NOTE

If the `ChoiceSet` contains an invalid set of `ChoiceCell`s, presenting the `ChoiceSet` will fail. 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
- Listeners: You must implement this listener interface 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 `GridView`) or a list (like a `RecyclerView`). If you are using tiles, it's recommended to use artworks on each item.

```
ChoiceSet choiceSet = new ChoiceSet("ChoiceSet Title", Arrays.asList(cell,
fullCell), new ChoiceSetSelectionListener() {
    @Override
    public void onChoiceSelected(ChoiceCell choiceCell, TriggerSource
triggerSource, int rowIndex) {
        // 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 that was passed.
        // <#handle selection#>
    }

    @Override
    public void onError(String error) {
        // <#handle error#>
    }
});
```

PRESENTING THE MENU WITH A MODE

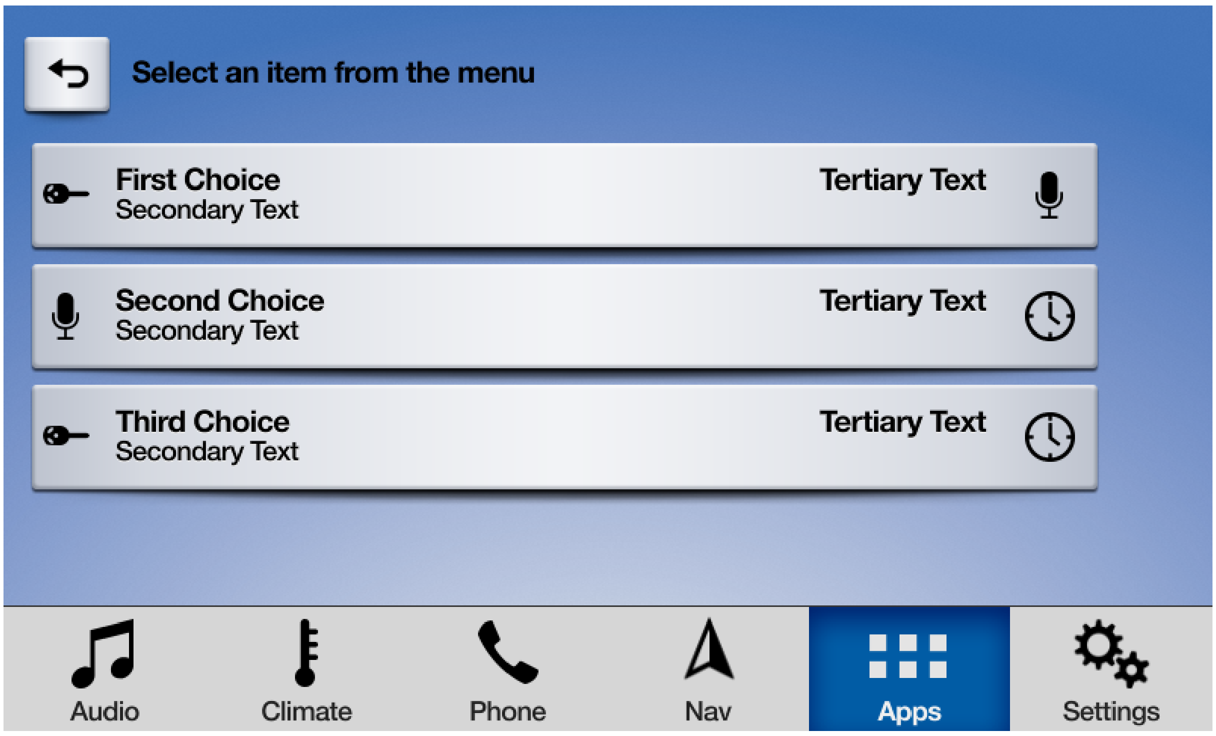
Finally, you will present the menu. When you do so, you must choose a `mode` to present it in. If you have no `vrCommands` on the choice cell you should choose `manualOnly`. If `vrCommands` are available, you may choose `voiceRecognitionOnly` or `both`.

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`.

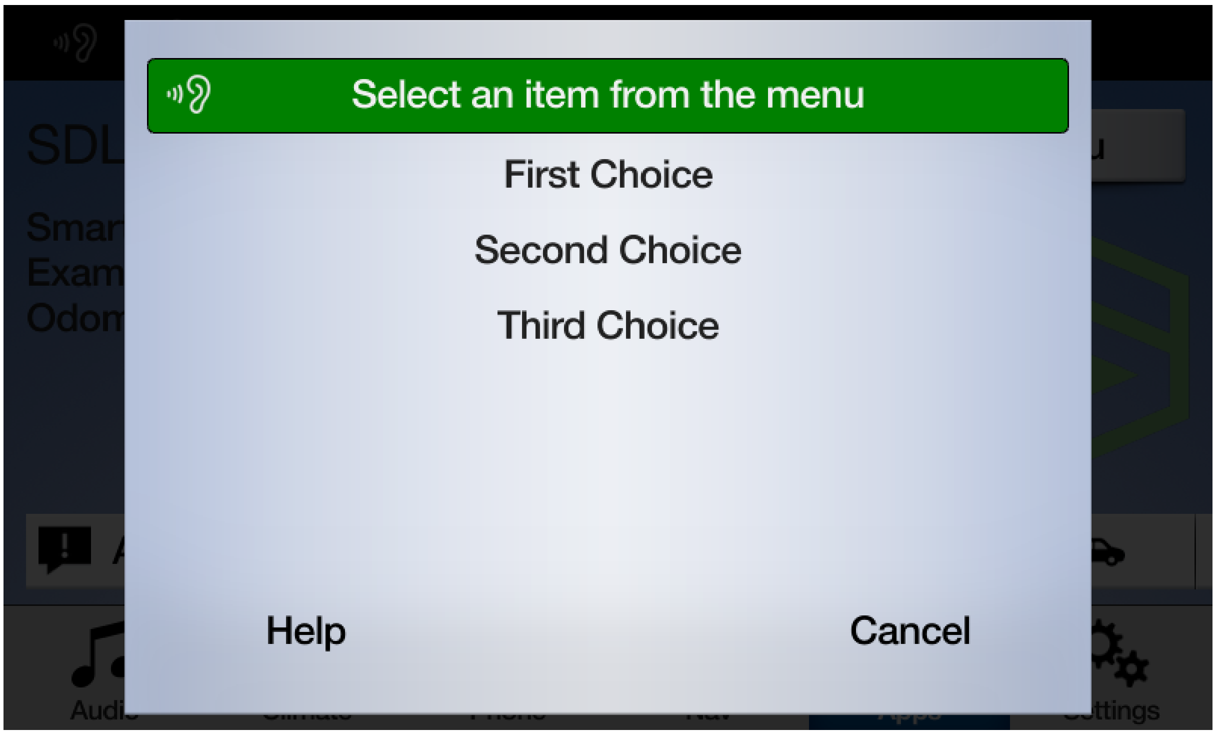
INTERACTION MODE	DESCRIPTION
Manual only	Interactions occur only through the display
VR only	Interactions occur only through text-to-speech and voice recognition
Both	Interactions can occur both manually or through VR

MENU - MANUAL ONLY MODE


Select an item from the menu

First Choice Secondary Text	Tertiary Text	
Second Choice Secondary Text	Tertiary Text	
Third Choice Secondary Text	Tertiary Text	

MENU - VOICE ONLY MODE

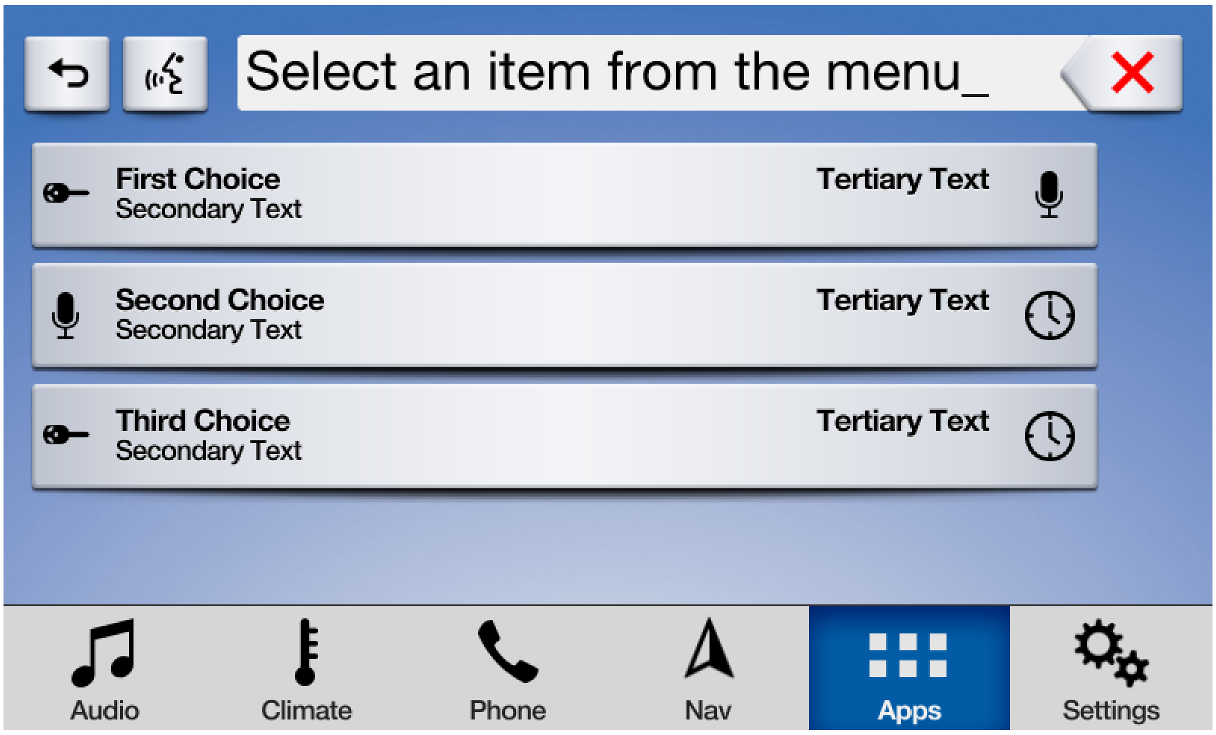


```
sdlManager.getScreenManager().presentChoiceSet(choiceSet,
InteractionMode.MANUAL_ONLY);
```

Presenting a Searchable Menu

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

MENU WITH SEARCH



```
sdlManager.getScreenManager().presentSearchableChoiceSet(choiceSet,
InteractionMode.MANUAL_ONLY, keyboardListener);
```

Deleting Cells

You can discover cells that have been preloaded on `sdlManager.getScreenManager().getPreloadedChoices()`. 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.

```
sdlManager.getScreenManager().deleteChoices(<List of choices to delete>);
```

Dismissing the Popup Menu (RPC v6.0+)

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

NOTE

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

```
choiceSet.cancel();
```

Using RPCs

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

Popup Keyboards

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

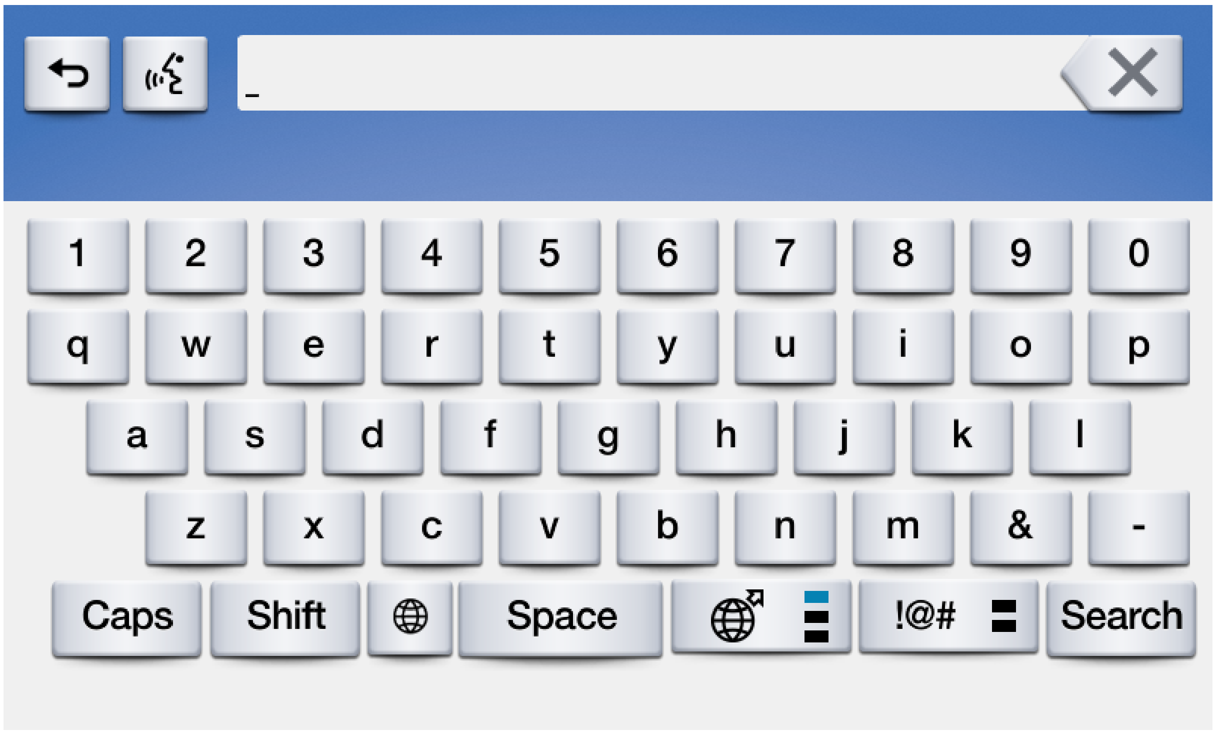
Presenting a Keyboard

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

NOTE

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

KEYBOARD SEARCH



```
int cancelId = sdlManager.getScreenManager().presentKeyboard("Initial text", null,
keyboardListener);
```

Implementing the Keyboard Listeners

Using the `KeyboardListener` involves implementing five methods:

```
KeyboardListener keyboardListener = new KeyboardListener() {
    @Override
    public void onUserDidSubmitInput(String inputText, KeyboardEvent event) {
        switch (event) {
            case ENTRY_VOICE:
                // <#The user decided to start voice input, you should start an
AudioPassThru session if supported#>
                break;
            case ENTRY_SUBMITTED:
                // <#The user submitted some text with the keyboard#>
                break;
            default:
                break;
        }
    }

    @Override
    public void onKeyboardDidAbortWithReason(KeyboardEvent event) {
        switch (event) {
            case ENTRY_CANCELLED:
                // <#The user cancelled the keyboard interaction#>
                break;
            case ENTRY_ABORTED:
                // <#The system aborted the keyboard interaction#>
                break;
            default:
                break;
        }
    }

    @Override
    public void updateAutocompleteWithInput(String currentInputText,
KeyboardAutocompleteCompletionListener
keyboardAutocompleteCompletionListener) {
        // <#Check the input text and return a list of autocomplete results#>
        //
keyboardAutocompleteCompletionListener.onUpdatedAutoCompleteList(<#String
 results to be displayed#>]);
    }

    @Override
    public void updateCharacterSetWithInput(String currentInputText,
KeyboardCharacterSetCompletionListener
keyboardCharacterSetCompletionListener) {
        // <#Check the input text and return a set of characters to allow the user to
enter#>
    }

    @Override
    public void onKeyboardDidSendEvent(KeyboardEvent event, String
currentInputText) {
        // <#This is sent upon every event, such as keypresses, cancellations, and
aborting#>
    }
};
```

Dismissing the Keyboard (RPC v6.0+)

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

NOTE

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

```
sdlManager.getScreenManager().dismissKeyboard(cancelId);
```

Using RPCs

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

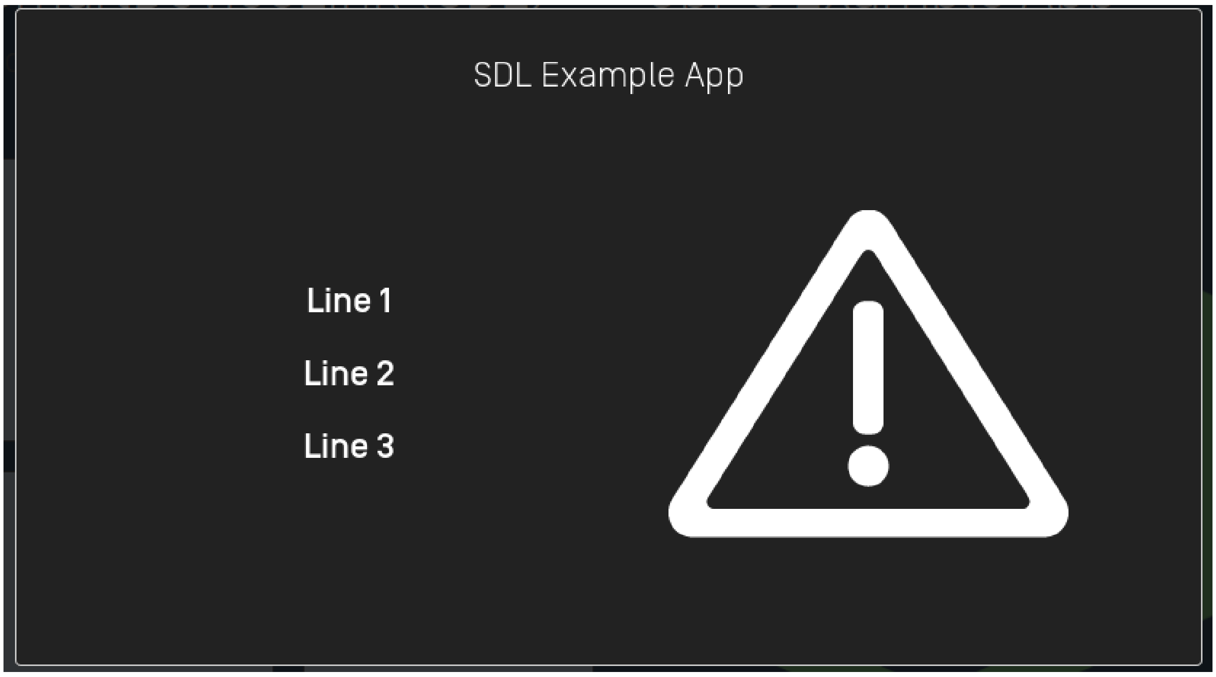
Alerts

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.

Alert Layouts

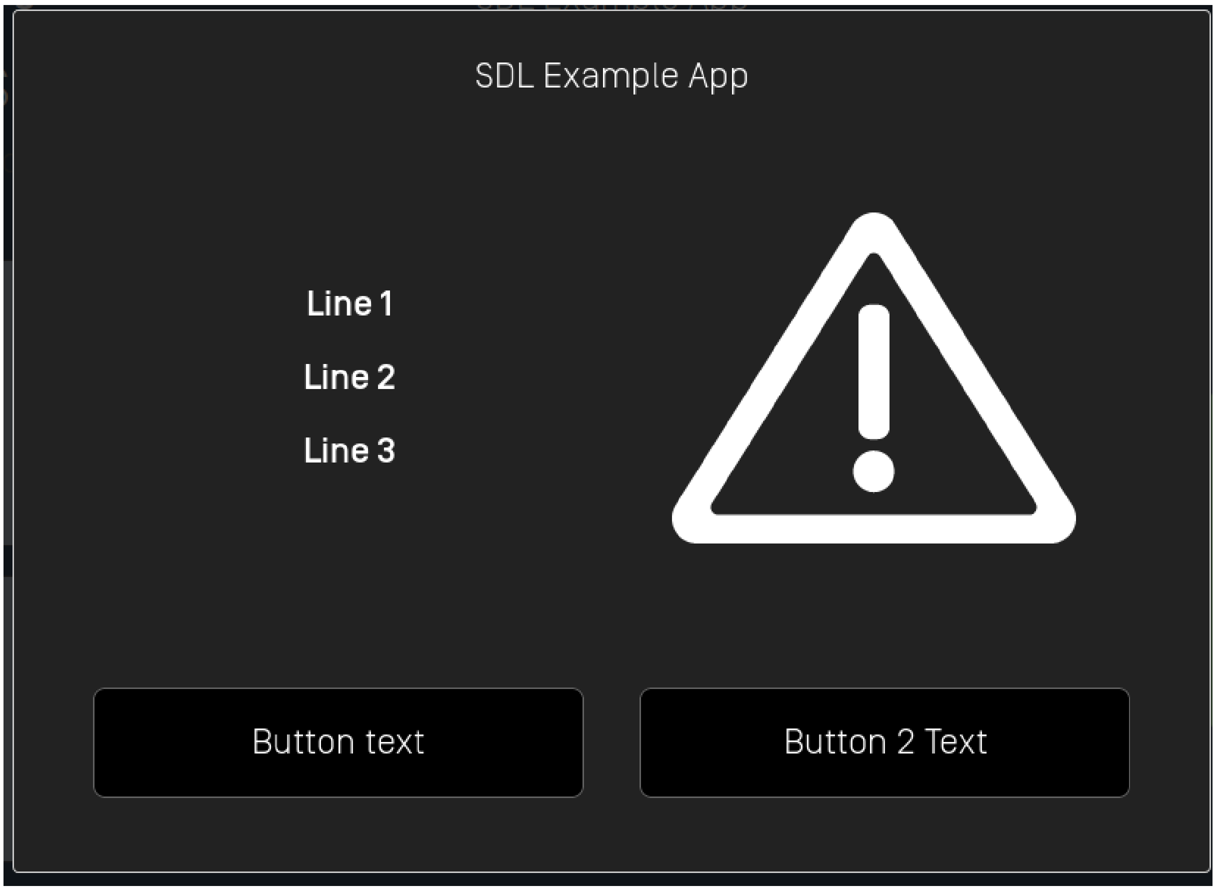
Alert With No Soft Buttons



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

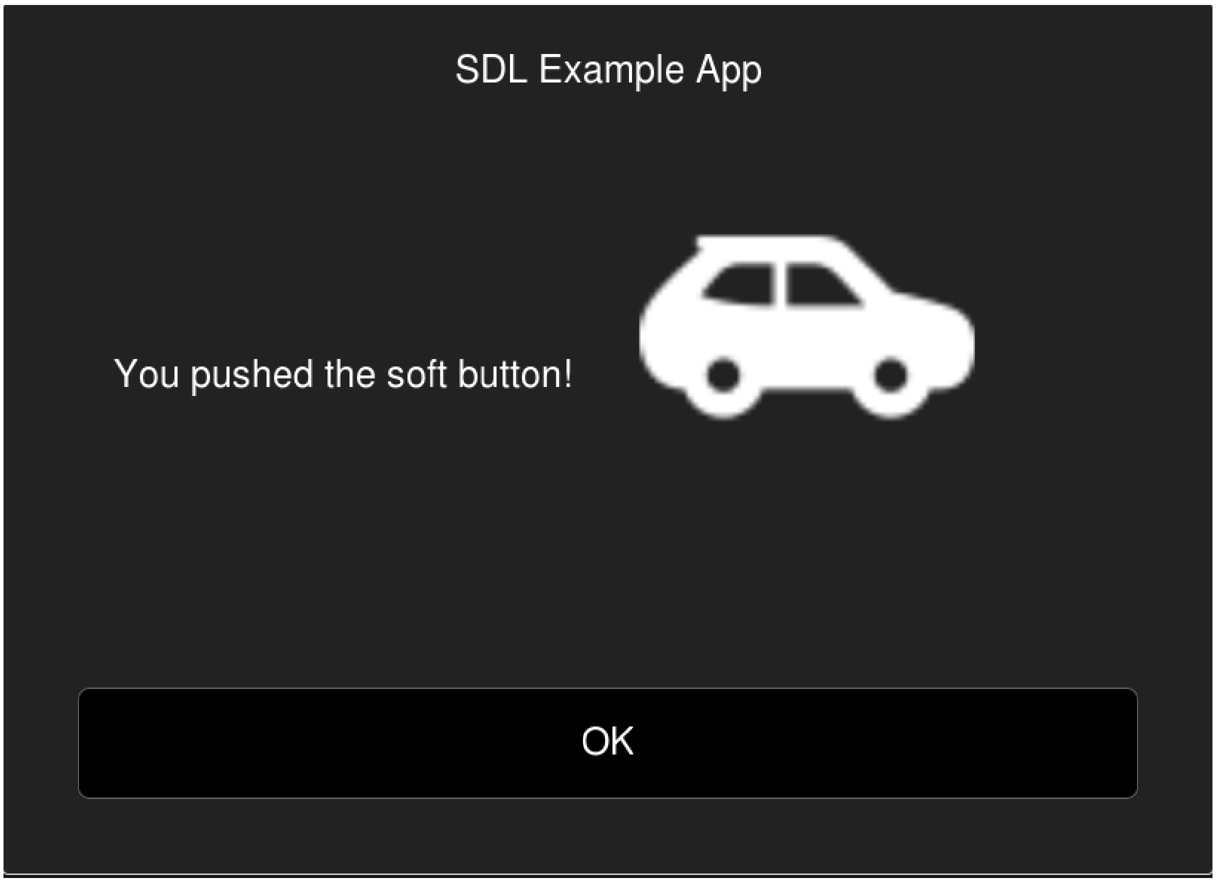
```
Alert alert = new Alert();
alert.setAlertText1("Line 1");
alert.setAlertText2("Line 2");
alert.setAlertText3("Line 3");
alert.setCancelID(<#Integer>);
```

Buttons

```
// Soft buttons
final int softButtonId = 123; // Set it to any unique ID
SoftButton okButton = new SoftButton(SoftButtonType.SBT_TEXT, softButtonId);
okButton.setText("OK");

// Set the softbuttons(s) to the alert
alert.setSoftButtons(Collections.singletonList(okButton));

// This listener is only needed once, and will work for all of soft buttons you send
with your alert
sdlManager.addOnRPCNotificationListener(FunctionID.ON_BUTTON_PRESS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPress = (OnButtonPress) notification;
        if (onButtonPress.getCustomButtonName() == softButtonId){
            Log.i(TAG, "Ok button pressed");
        }
    }
});
```

Alert Icon

An alert can include a custom or static (built-in) image that will be displayed within the alert. Before you add the image to the alert make sure the image is uploaded to the head unit using the FileManager. If the image is already uploaded, you can set the `alertIcon` property.



You pushed the soft button!

```
alert.setAlertIcon(new Image(<#artworkName#>, ImageType.DYNAMIC));
```

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.

```
alert.setDuration(5000);
```

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.

```
alert.setProgressIndicator(true);
```

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

```
alert.setTtsChunks(TTSChunkFactory.createSimpleTTSChunks("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.

```
TTSChunk ttsChunk = new TTSChunk(sdlFile.getName(),
SpeechCapabilities.FILE);
alert.setTtsChunks(Collections.singletonList(ttsChunk));
```

Play Tone

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

```
alert.setPlayTone(true);
```

Showing the Alert

```
// Handle RPC response
alert.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
      if (response.getSuccess()){
        Log.i(TAG, "Alert was shown successfully");
      }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
      Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(alert);
```

Dismissing the Alert (RPC v6.0+)

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

NOTE

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

Please note that canceling the alert will only dismiss the displayed alert. If you have set the `ttsChunk` property, the speech will play in its entirety even when the displayed alert has been dismissed. If you know you will cancel an alert, consider setting a short `ttsChunk` like "searching" instead of "searching for coffee shops, please wait."

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

Dismissing a Specific Alert

```
// `cancelID` is the ID that you assigned when creating and sending the alert
CancelInteraction cancelInteraction = new
CancelInteraction(FunctionID.ALERT.getId(), cancelID);
cancelInteraction.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            Log.i(TAG, "Alert was dismissed successfully");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(cancelInteraction);
```

Dismissing the Current Alert

```
CancelInteraction cancelInteraction = new
CancelInteraction(FunctionID.ALERT.getId());
cancelInteraction.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            Log.i(TAG, "Alert was dismissed successfully");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(cancelInteraction);
```

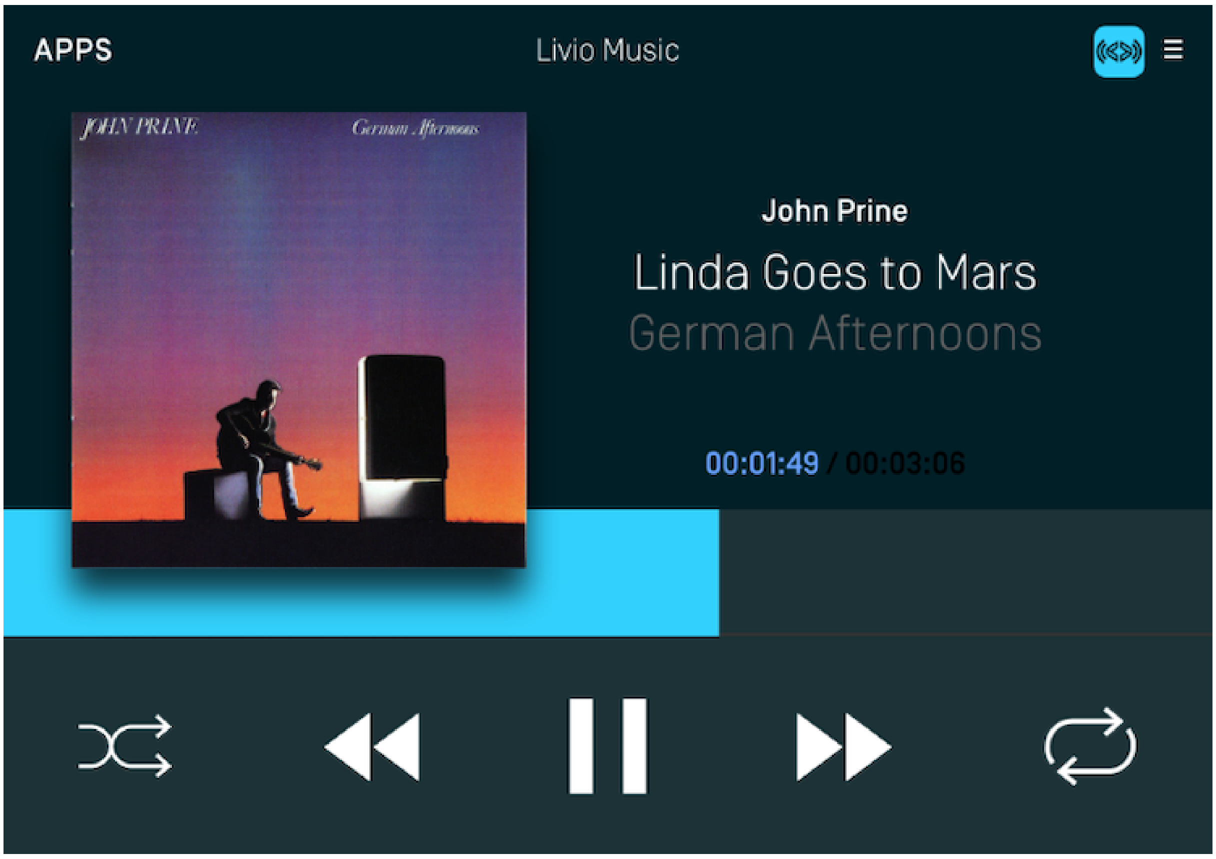
Media Clock

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"

```
SetMediaClockTimer mediaClock = new
SetMediaClockTimer().countUpFromStartTimeInterval(30, 253,
AudioStreamingIndicator.PAUSE);
sdlManager.sendRPC(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`.

```
SetMediaClockTimer mediaClock = new
SetMediaClockTimer().countDownFromStartTimeInterval(600, 0,
AudioStreamingIndicator.PAUSE);
sdlManager.sendRPC(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.

```
SetMediaClockTimer mediaClock = new
SetMediaClockTimer().pauseWithPlayPauseIndicator(AudioStreamingIndicator.PL

sdlManager.sendRPC(mediaClock);
```

```
SetMediaClockTimer mediaClock = new
SetMediaClockTimer().resumeWithPlayPauseIndicator(AudioStreamingIndicator.P

sdlManager.sendRPC(mediaClock);
```

```
SetMediaClockTimer mediaClock = new
SetMediaClockTimer().updatePauseWithNewStartTimeInterval(60, 240,
AudioStreamingIndicator.PLAY);
sdlManager.sendRPC(mediaClock);
```

Clearing the Timer

Clearing the timer removes it from the screen.

```
SetMediaClockTimer mediaClock = new
SetMediaClockTimer().clearWithPlayPauseIndicator(AudioStreamingIndicator.PLA

sdlManager.sendRPC(mediaClock);
```

Updating the Audio Indicator

The audio indicator is, essentially, the play / pause button. As of library v.4.7, 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 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 `Slider` creates a full screen or pop-up overlay (depending on platform) that a user can control. There are two main `Slider` layouts, one with a static footer and one with a dynamic footer.

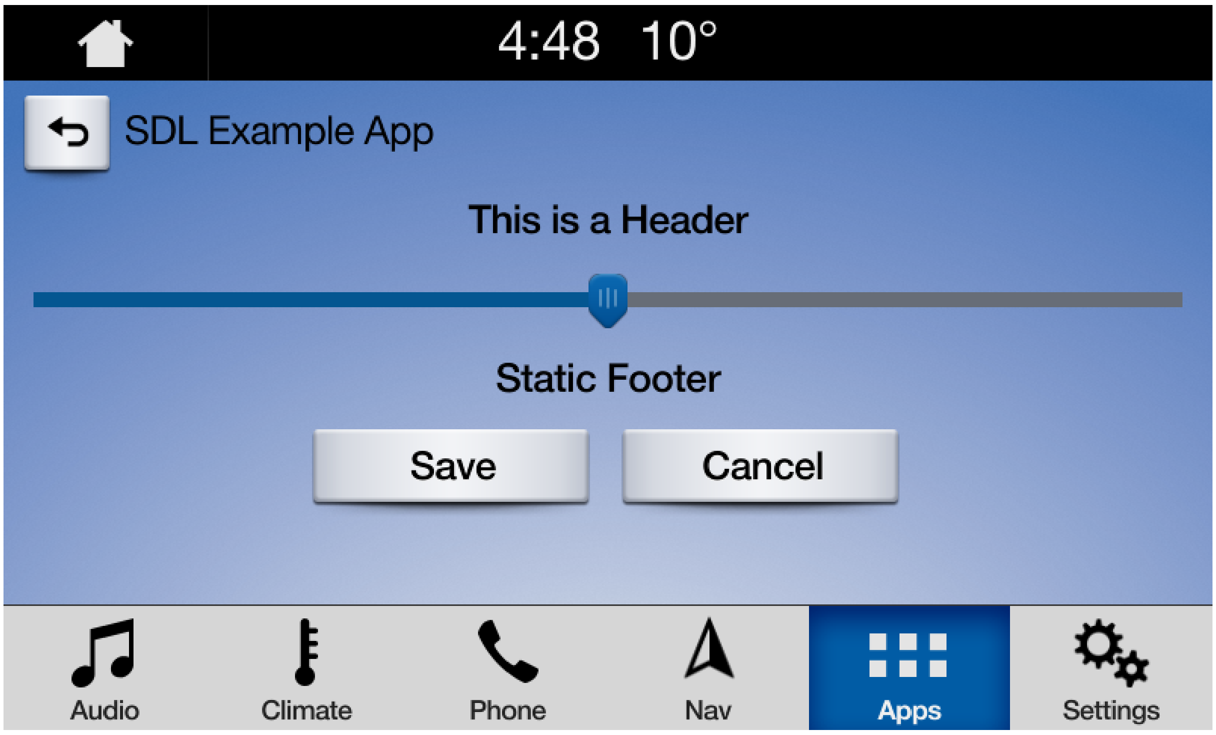
NOTE

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

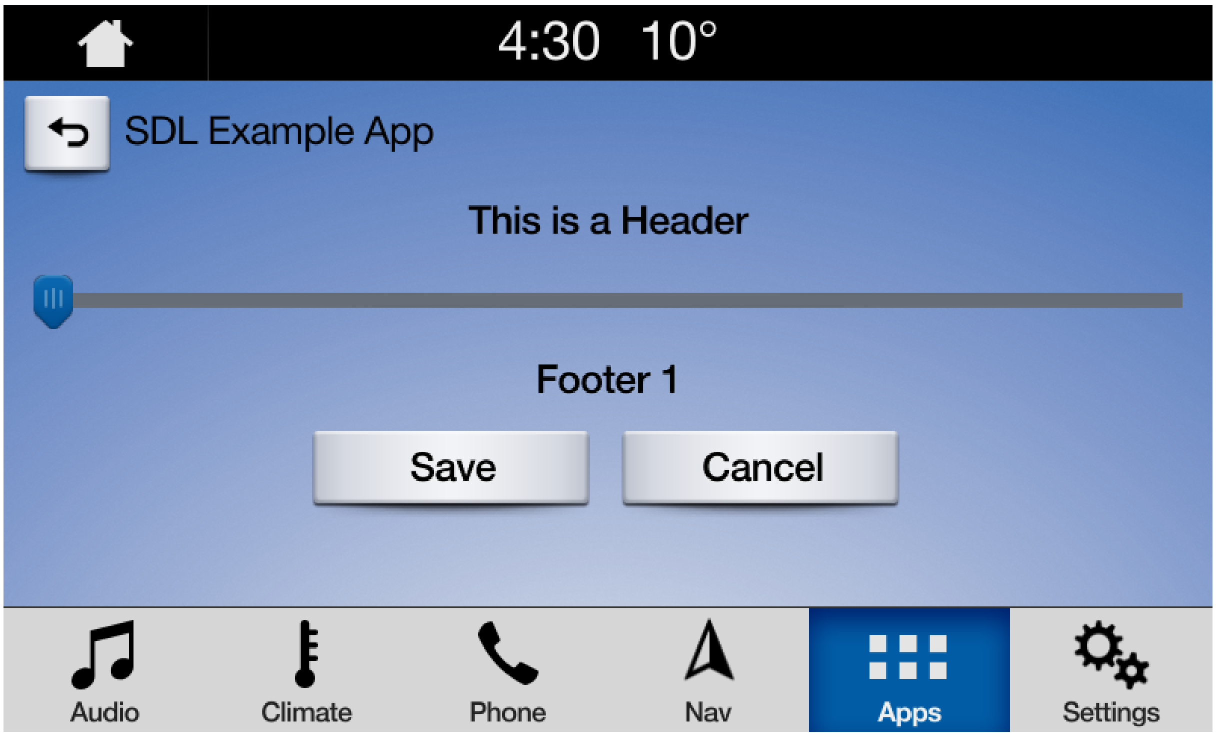
Slider

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

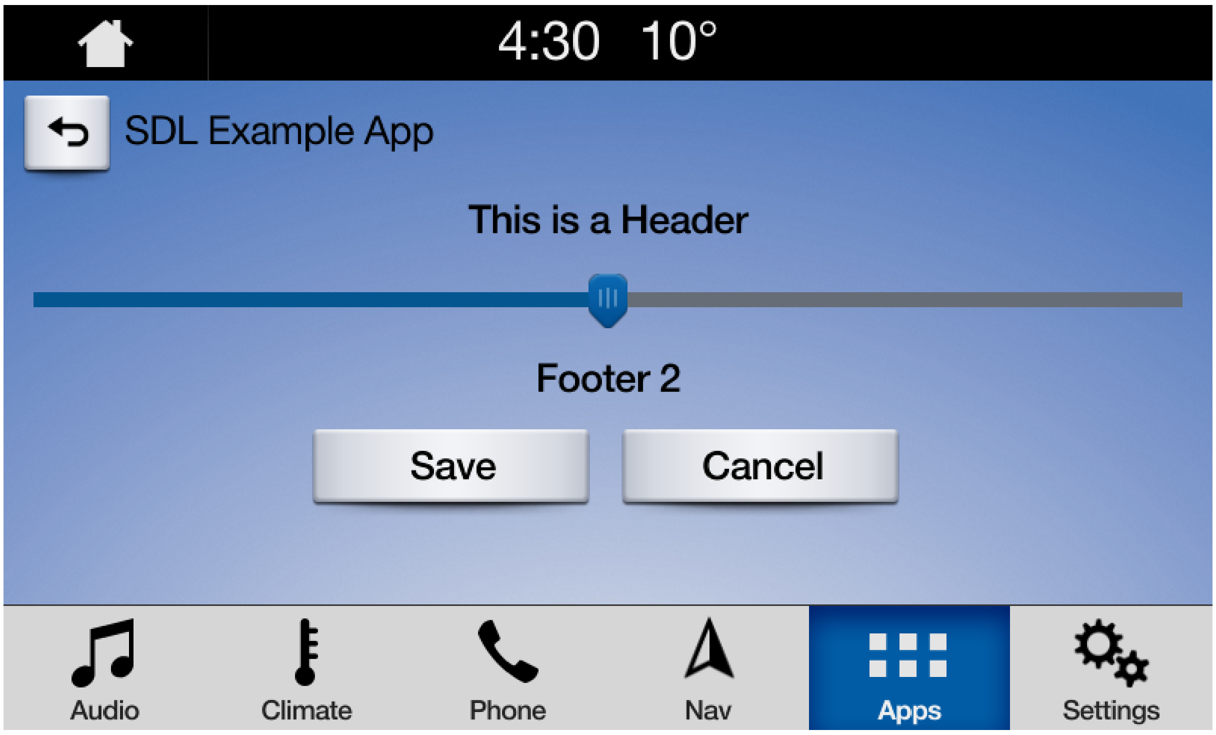
Slider UI



DYNAMIC SLIDER IN POSITION 1



DYNAMIC SLIDER IN POSITION 2



Creating the Slider

```
Slider slider = new Slider();
```

Ticks

The number of selectable items on a horizontal axis.

```
// Must be a number between 2 and 26
slider.setNumTicks(5);
```

Position

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

```
// Must be a number between 1 and 26
slider.setPosition(1);
```

Header

The header to display.

```
// Max length 500 chars
slider.setSliderHeader("This is a Header");
```

Static Footer

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

```
// Max length 500 chars
slider.setSliderFooter(Collections.singletonList("Static Footer"));
```

Dynamic Footer

This type of footer will have a different message displayed for each position of the slider. The footer is an optional paramater. 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 `null` to have no footer at all.

```
// Array length 1 - 26, Max length 500 chars
slider.setSliderFooter(Arrays.asList("Footer 1","Footer 2","Footer 3"));
```

Cancel ID

An ID for this specific slider to allow cancellation through the `CancelInteraction` RPC.

```
slider.setCancelID(45);
```

Show the Slider

```
slider.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        SliderResponse sliderResponse = (SliderResponse) response;
        Log.i(TAG, "Slider Position Set: " + sliderResponse.getSliderPosition());
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: " + resultCode + " | Info: " + info);
    }
});
sdlManager.sendRPC(slider);
```

Dismissing a Slider (RPC v6.0+)

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

NOTE

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

Dismissing a Specific Slider

```
// `cancelID` is the ID that you assigned when creating the slider
CancelInteraction cancelInteraction = new
CancelInteraction(FunctionID.SLIDER.getId(), cancelID);
cancelInteraction.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            Log.i(TAG, "Slider was dismissed successfully");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(cancelInteraction);
```

Dismissing the Current Slider

```
CancelInteraction cancelInteraction = new
CancelInteraction(FunctionID.SLIDER.getId());
cancelInteraction.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            Log.i(TAG, "Slider was dismissed successfully");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(cancelInteraction);
```

Scrollable Message

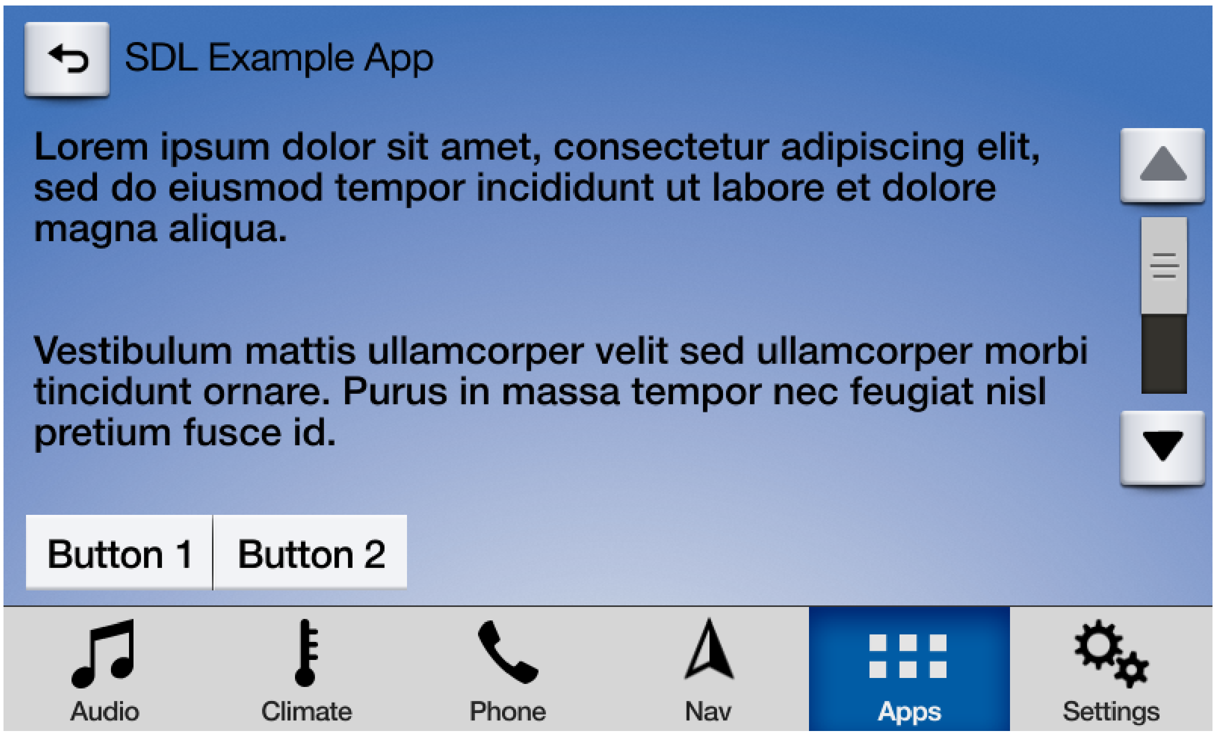
A `ScrollableMessage` creates an overlay containing a large block of formatted text that can be scrolled. `ScrollableMessage` contains a body of text, a message timeout, and up to 8 soft buttons depending on head unit. You must check the `sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().getSoftButtonCapabilities()`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

You simply create a `ScrollableMessage` RPC request and send it to display the 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



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.

Creating the Scrollable Message

```
// Create Message To Display
String 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 SoftButtons
SoftButton softButton1 = new SoftButton(SoftButtonType.SBT_TEXT, 0);
softButton1.setText("Button 1");

SoftButton softButton2 = new SoftButton(SoftButtonType.SBT_TEXT, 1);
softButton2.setText("Button 2");

// Create SoftButton Array
List<SoftButton> softButtonList = Arrays.asList(softButton1, softButton2);

// Create ScrollableMessage Object
ScrollableMessage scrollableMessage = new ScrollableMessage();
scrollableMessage.setScrollableMessageBody(scrollableMessageText);
scrollableMessage.setTimeout(50000);
scrollableMessage.setSoftButtons(softButtonList);

// Set cancelId
scrollableMessage.setCancelID(<#Integer>);

// Send the scrollable message
sdlManager.sendRPC(scrollableMessage);
```

To listen for `OnButtonPress` events for `SoftButton`s, we need to add a listener that listens for their Id's:

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_BUTTON_PRESS, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnButtonPress onButtonPress = (OnButtonPress) notification;
        switch (onButtonPress.getCustomButtonName()){
            case 0:
                Log.i(TAG, "Button 1 Pressed");
                break;
            case 1:
                Log.i(TAG, "Button 2 Pressed");
                break;
        }
    }
});
```

Dismissing a Scrollable Message (RPC v6.0+)

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

NOTE

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

Dismissing a Specific Scrollable Message

```
// `cancelID` is the ID that you assigned when creating and sending the alert
CancelInteraction cancelInteraction = new
CancelInteraction(FunctionID.SCROLLABLE_MESSAGE.getId(), cancelID);
cancelInteraction.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            Log.i(TAG, "Scrollable message was dismissed successfully");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(cancelInteraction);
```

Dismissing the Current Scrollable Message

```
CancelInteraction cancelInteraction = new
CancelInteraction(FunctionID.SCROLLABLE_MESSAGE.getId());
cancelInteraction.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            Log.i(TAG, "Scrollable message was dismissed successfully");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(cancelInteraction);
```

Customizing the Template

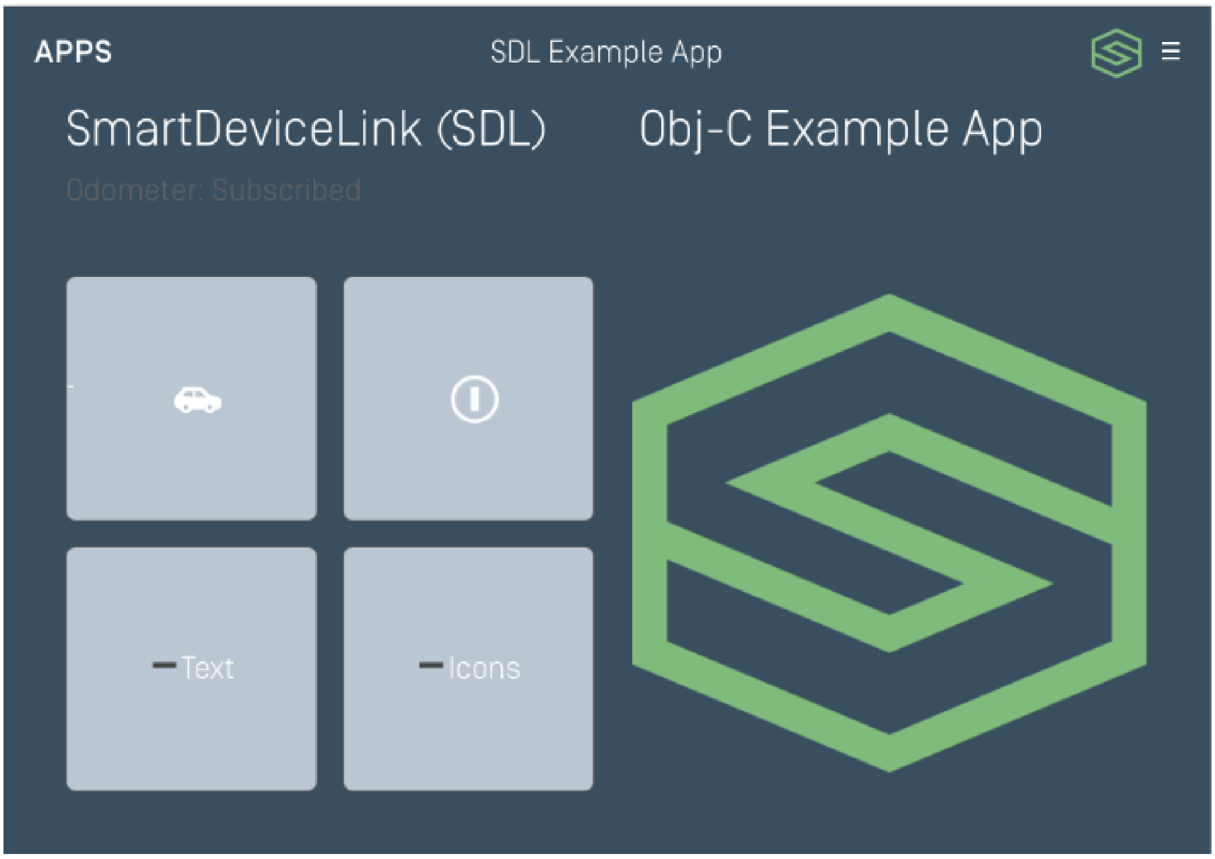
You have the ability to customize the look and feel of the template. How much customization is available depends on the RPC version of the head unit you are connected with as well as the design of the HMI.

Customizing Template Colors (RPC v5.0+)

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

Customizing the Default Layout

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


APPS

SDL Example App

SmartDeviceLink (SDL) Obj-C Example App

Odometer: Subscribed

```
// Set color schemes
RGBColor green = new RGBColor(126, 188, 121);
RGBColor white = new RGBColor(249, 251, 254);
RGBColor grey = new RGBColor(186, 198, 210);
RGBColor darkGrey = new RGBColor(57, 78, 96);

TemplateColorScheme dayColorScheme = new TemplateColorScheme();
dayColorScheme.setBackgroundColor(white);
dayColorScheme.setPrimaryColor(green);
dayColorScheme.setSecondaryColor(grey);
builder.setDayColorScheme(dayColorScheme);

TemplateColorScheme nightColorScheme = new TemplateColorScheme();
nightColorScheme.setBackgroundColor(white);
nightColorScheme.setPrimaryColor(green);
nightColorScheme.setSecondaryColor(darkGrey);
builder.setDayColorScheme(nightColorScheme);
```

NOTE

You may change the template coloring in the `lifecycleConfiguration` and the `SetDisplayLayout`, if connecting to a head unit with RPC v5.0+, or with the `Show` request if connecting to RPC v6.0+. You may only change the template coloring once per template; that is, you cannot call `SetDisplayLayout` or `Show` for the template you are already on and expect the color scheme to update.

Customizing Future Layouts

You can change the template color scheme when you change layouts in the `SetDisplayLayout` (any RPC version) or `Show` (RPC v6.0+) request.

```
// Set color schemes
RGBColor green = new RGBColor(126, 188, 121);
RGBColor white = new RGBColor(249, 251, 254);
RGBColor grey = new RGBColor(186, 198, 210);
RGBColor darkGrey = new RGBColor(57, 78, 96);

TemplateColorScheme dayColorScheme = new TemplateColorScheme();
dayColorScheme.setBackgroundColor(white);
dayColorScheme.setPrimaryColor(green);
dayColorScheme.setSecondaryColor(grey);
builder.setDayColorScheme(dayColorScheme);

TemplateColorScheme nightColorScheme = new TemplateColorScheme();
nightColorScheme.setBackgroundColor(white);
nightColorScheme.setPrimaryColor(green);
nightColorScheme.setSecondaryColor(darkGrey);

SetDisplayLayout setDisplayLayout = new
SetDisplayLayout(PredefinedLayout.GRAPHIC_WITH_TEXT.toString());
setDisplayLayout.setDayColorScheme(dayColorScheme);
setDisplayLayout.setNightColorScheme(nightColorScheme);
setDisplayLayout.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            // Success
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        // Handle error
    }
});
sdlManager.sendRPC(setDisplayLayout);
```

Customizing the Menu Title and Icon

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

```
SetGlobalProperties setGlobalProperties = new SetGlobalProperties();
setGlobalProperties.setMenuTitle("customTitle");
// The image must be uploaded before referencing the image name here
setGlobalProperties.setMenuIcon(<#Image#>);
setGlobalProperties.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()){
            // Success
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        // Handle error
    }
});
sdlManager.sendRPC(setGlobalProperties);
```

Customizing the Keyboard

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

Setting Keyboard Properties

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

```
KeyboardProperties keyboardProperties = new KeyboardProperties();
keyboardProperties.setLanguage(Language.HE_IL); // Set to Israeli Hebrew
keyboardProperties.setKeyboardLayout(KeyboardLayout.AZERTY); // Set to
AZERTY

sdlManager.getScreenManager().setKeyboardConfiguration(keyboardProperties);
```

Other Properties

While there are other keyboard properties available on `KeyboardProperties`, these will be overridden by the screen manager. The `keypressMode` must be a specific configuration for the screen manager's callbacks to work properly. The `limitedCharacterList`, `autoCompleteText`, and `autoCompleteList` will be set on a per-keyboard basis when calling `sdlManager.getScreenManager.presentKeyboard(...)`, should custom keyboard properties be set.

Customizing Help Prompts

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

Configuring the Help Menu

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

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

```
SetGlobalProperties setGlobalProperties = new SetGlobalProperties();
setGlobalProperties.setVrHelpTitle("What Can I Say?");

VrHelpItem item1 = new VrHelpItem("Show Artists", 1);
item1.setImage(<#image#>); // a previously uploaded image or null

VrHelpItem item2 = new VrHelpItem("Show Albums", 2);
item2.setImage(<#image#>); // a previously uploaded image or null

setGlobalProperties.setVrHelp(Arrays.asList(item1, item2));
setGlobalProperties.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // The help menu is updated
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(setGlobalProperties);
```

Configuring the Help Prompt

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

```
SetGlobalProperties setGlobalProperties = new SetGlobalProperties();
setGlobalProperties.setHelpPrompt(TTSChunkFactory.createSimpleTTSChunks("
 custom help prompt"));
setGlobalProperties.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // The help prompt is updated
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(setGlobalProperties);
```

Configuring the Timeout Prompt

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

```
SetGlobalProperties setGlobalProperties = new SetGlobalProperties();
setGlobalProperties.setTimeoutPrompt(TTSChunkFactory.createSimpleTTSChunk
 custom help prompt"));
setGlobalProperties.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // The timeout prompt is updated
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(setGlobalProperties);
```

Clearing Help Menu and Prompt Customizations

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

```
// Reset the help menu
ResetGlobalProperties resetGlobalProperties = new
ResetGlobalProperties(Arrays.asList(GlobalProperty.VRHELPITEMS,
GlobalProperty.VRHELPTITLE));

// Reset the menu icon and title
ResetGlobalProperties resetGlobalProperties = new
ResetGlobalProperties(Arrays.asList(GlobalProperty.MENUICON,
GlobalProperty.MENUNAME));

// Reset spoken prompts
ResetGlobalProperties resetGlobalProperties = new
ResetGlobalProperties(Arrays.asList(GlobalProperty.HELPPROMPT,
GlobalProperty.TIMEOUTPROMPT));

// To send any one of these, use the typical format:
resetGlobalProperties.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // The global properties are reset
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(resetGlobalProperties);
```

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 `Speak` request and the head unit's text-to-speech (TTS) engine will produce synthesized speech from your provided text.

When using the `Speak` 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 `sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.SPEECH)` 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

```
TTSChunk ttsChunk = new TTSChunk("hello", SpeechCapabilities.TEXT);
List<TTSChunk> ttsChunkList = Collections.singletonList(ttsChunk);
Speak speak = new Speak(ttsChunkList);
```

SAPI Phonemes Phrase

```
TTSChunk ttsChunk = new TTSChunk("h eh - l ow 1",
SpeechCapabilities.SAPI_PHONEMES);
List<TTSChunk> ttsChunkList = Collections.singletonList(ttsChunk);
Speak speak = new Speak(ttsChunkList);
```

Sending the Speak Request

```
speak.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        SpeakResponse speakResponse = (SpeakResponse) response;
        if (!speakResponse.getSuccess()){
            switch (speakResponse.getResultCode()){
                case DISALLOWED:
                    Log.i(TAG, "The app does not have permission to use the speech
request");
                    break;
                case REJECTED:
                    Log.i(TAG, "The request was rejected because a higher priority
request is in progress");
                    break;
                case ABORTED:
                    Log.i(TAG, "The request was aborted by another higher priority
request");
                    break;
                default:
                    Log.i(TAG, "Some other error occurred");
            }
            return;
        }
        Log.i(TAG, "Speech was successfully spoken");
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        Log.i(TAG, "onError: " + info);
    }
});
sdlManager.sendRPC(speak);
```

Playing Audio Indications

As of library v.4.7 and SDL Core v.5.0+, you can pass an uploaded audio file's name to `TTSChunk`, 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 `FileManager`.

```
SdlFile audioFile = new SdlFile("Audio file name", FileType.AUDIO_MP3,
Uri.parse("File Location"), true);
sdlManager.getFileManager().uploadFile(audioFile, new CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

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 `TTSChunk` referring to the file's name.

```
Alert alert = new Alert();
alert.setAlertText1("Alert Text 1");
alert.setAlertText2("Alert Text 2");
alert.setDuration(5000);
alert.setTtsChunks(Arrays.asList(new TTSChunk("Audio file name",
SpeechCapabilities.FILE)));
```

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 `VoiceCommand` objects to the `voiceCommands` array on the screen manager.

```
VoiceCommand voiceCommand = new
VoiceCommand(Collections.singletonList("Command One"), new
VoiceCommandSelectionListener() {
    @Override
    public void onVoiceCommandSelected() {
        // <#Handle the VoiceCommand's Selection#>
    }
});

sdlManager.getScreenManager().setVoiceCommands(Collections.singletonList(v
```

Using RPCs

If you wish to do this without the aid of the screen manager, you can create `AddCommand` 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 `PerformAudioPassThru` 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 `EndAudioPassThru`.

NOTE

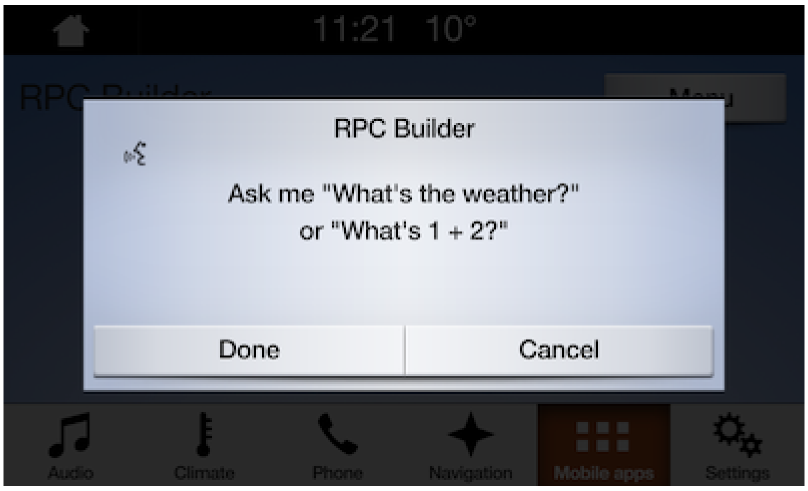
SDL does not support an open microphone. However, SDL is working on wake-word support in the future. You may implement a voice command and start an audio pass thru session when that voice command occurs.

Starting Audio Capture

To initiate audio capture, first construct a `PerformAudioPassThru` 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 `sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.AUDIO_PASSTHROUGH)` 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

```
PerformAudioPassThru performAPT = new PerformAudioPassThru();
performAPT.setAudioPassThruDisplayText1("Ask me \"What's the weather?\"");
performAPT.setAudioPassThruDisplayText2("or \"What's 1 + 2?\"");

performAPT.setInitialPrompt(TTSChunkFactory.createSimpleTTSChunks("Ask
me What's the weather? or What's 1 plus 2?"));
performAPT.setSamplingRate(SamplingRate._22KHZ);
performAPT.setMaxDuration(7000);
performAPT.setBitsPerSample(BitsPerSample._16_BIT);
performAPT.setAudioType(AudioType.PCM);
performAPT.setMuteAudio(false);

sdlManager.sendRPC(performAPT);
```



RPC Builder

Ask me "What's the weather?"
or "What's 1 + 2?"

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 observe the `OnAudioPassThru` notification.

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.

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_AUDIO_PASS_THRU,
new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnAudioPassThru onAudioPassThru = (OnAudioPassThru) notification;
        byte[] dataRcvd = onAudioPassThru.getAPTData();
        processAPTData(dataRcvd); // Do something with audio data
    }
});
```

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

`PerformAudioPassThru` 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 `EndAudioPassThru` RPC. You will receive a `resultCode` of `SUCCESS`, and should handle the audio pass thru as though it was successful.

```
EndAudioPassThru endAPT = new EndAudioPassThru();
sdlManager.sendRPC(endAPT);
```

Handling the Response

To process the response received from an ended audio capture, monitor the `PerformAudioPassThruResponse` by adding a listener to the `PerformAudioPassThru` RPC before sending it. If the response has a successful result, all of the audio data for the passthrough has been received and is ready for processing.

```
performAPT.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        Result result = response.getResultCode();

        if(result.equals(Result.SUCCESS)){
            // We can use the data
        }else{
            // Cancel any usage of the data
            Log.e("SdlService", "Audio pass thru attempt failed.");
        }
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
```

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 listener that is specific to them, the `OnMultipleRequestListener`. This listener will provide more information than the normal `OnRPCResponseListener`.

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.

```
SubscribeButton subscribeButtonLeft = new
SubscribeButton(ButtonName.SEEKLEFT);
SubscribeButton subscribeButtonRight = new
SubscribeButton(ButtonName.SEEKRIGHT);
sdlManager.sendRPCs(Arrays.asList(subscribeButtonLeft, subscribeButtonLeft),
new OnMultipleRequestListener() {
    @Override
    public void onUpdate(int remainingRequests) {

    }

    @Override
    public void onFinished() {

    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {

    }

    @Override
    public void onResponse(int correlationId, RPCResponse response) {

    }
});
```

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 `PerformInteraction` RPC can only be sent after the `CreateInteractionChoiceSet` RPC has been registered by Core, which is why the requests must be sent sequentially.

```
int choiceId = 111, choiceSetId = 222;
Choice choice = new Choice(choiceId, "Choice title");
CreateInteractionChoiceSet createInteractionChoiceSet = new
CreateInteractionChoiceSet(choiceSetId, Collections.singletonList(choice));
PerformInteraction performInteraction = new PerformInteraction("Initial Text",
InteractionMode.MANUAL_ONLY, Collections.singletonList(choiceSetId));
sdlManager.sendSequentialRPCs(Arrays.asList(createInteractionChoiceSet,
performInteraction), new OnMultipleRequestListener() {
    @Override
    public void onUpdate(int i) {

    }

    @Override
    public void onFinished() {

    }

    @Override
    public void onError(int i, Result result, String s) {

    }

    @Override
    public void onResponse(int i, RPCResponse rpcResponse) {

    }
});
```

Retrieving Vehicle Data

You can use the `GetVehicleData` and `SubscribeVehicleData` 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
Cluster Mode Status	clusterModeStatus	Whether or not the power mode is active. The power mode qualification status: power mode undefined, power mode evaluation in progress, not defined, power mode ok. The car mode status: normal, factory, transport, or crash. The power mode status: key out, key recently out, key approved, post accessory, accessory, post ignition, ignition on, running, crank

VEHICLE DATA	PARAMETER NAME	DESCRIPTION
Device Status	deviceStatus	Contains information about the smartphone device. Is voice recognition on or off, has a bluetooth connection been established, is a call active, is the phone in roaming mode, is a text message available, the battery level, the status of the mono and stereo output channels, the signal level, the primary audio source, whether or not an emergency call is currently taking place
Driver Braking	driverBraking	The status of the brake pedal: yes, no, no event, fault, not supported
E-Call Information	eCallInfo	Information about the status of an emergency call
Electronic Parking Brake Status	electronicParkingBrakeStatus	The status of the electronic parking brake. Available states: closed, transition, open, drive active, fault

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 `GetVehicleData` RPC.

```
GetVehicleData vdRequest = new GetVehicleData();
vdRequest.setPrndl(true);
vdRequest.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            PRNDL prndl = ((GetVehicleDataResponse) response).getPrndl();
            Log.i("SdlService", "PRNDL status: " + prndl.toString());
        }else{
            Log.i("SdlService", "GetVehicleData was rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(vdRequest);
```

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, you should add a notification listener for the `OnVehicleData` notification:

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_VEHICLE_DATA, new
OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnVehicleData onVehicleDataNotification = (OnVehicleData) notification;
        if (onVehicleDataNotification.getPrndl() != null) {
            Log.i("SdlService", "PRNDL status was updated to: " +
onVehicleDataNotification.getPrndl());
        }
    }
});
```

Second, send the `SubscribeVehicleData` request:

```
SubscribeVehicleData subscribeRequest = new SubscribeVehicleData();
subscribeRequest.setPrndl(true);
subscribeRequest.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Log.i("SdlService", "Successfully subscribed to vehicle data.");
        }else{
            Log.i("SdlService", "Request to subscribe to vehicle data was rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(subscribeRequest);
```

Third, the `onNotified` method will be called when there is an update to the subscribed vehicle 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 `UnsubscribeVehicleData` RPC.

```
UnsubscribeVehicleData unsubscribeRequest = new UnsubscribeVehicleData();
unsubscribeRequest.setPrndl(true); // unsubscribe to PRNDL data
unsubscribeRequest.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Log.i("SdlService", "Successfully unsubscribed to vehicle data.");
        }else{
            Log.i("SdlService", "Request to unsubscribe to vehicle data was rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(unsubscribeRequest);
```

OEM-Specific Vehicle Data

OEM applications can access additional vehicle data published by their systems that is not available via the SDL vehicle data APIs. This data is accessed using the same SDL vehicle data RPCs, but instead of requesting a certain type of SDL-specified data, you must request data using a custom vehicle data name. The type of object returned is up to the OEM and must be parsed manually.

NOTE

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

Requesting One-Time OEM-Specific Vehicle Data

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

```
GetVehicleData vdRequest = new GetVehicleData();
vdRequest.setOEMCustomVehicleData("OEM-X-Vehicle-Data", true);
vdRequest.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if(response.getSuccess()){
            Object CustomData = ((GetVehicleDataResponse)
response).getOEMCustomVehicleData("OEM-X-Vehicle-Data");
        }else{
            Log.i("SdlService", "GetVehicleData was rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
  });
sdlManager.sendRPC(vdRequest);
```

Remote Control Vehicle Features

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

NOTE

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

Why Use Remote Control?

Consider the following scenarios:

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

Supported Modules

Currently, the remote control feature supports these modules:

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

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

CLIMATE

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

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

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

RADIO

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

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

SEAT

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Seat Heating Enabled	heatingEnabled	true, false	Get/Set/Notification	Indicates whether heating is enabled for a seat	Since v5.0
Seat Cooling Enabled	coolingEnabled	true, false	Get/Set/Notification	Indicates whether cooling is enabled for a seat	Since v5.0
Seat Heating level	heatingLevel	0-100%	Get/Set/Notification	Level of the seat heating	Since v5.0
Seat Cooling level	coolingLevel	0-100%	Get/Set/Notification	Level of the seat cooling	Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Seat Horizontal Position	horizontalPosition	0-100%	Get/Set/Notification	Adjust a seat forward/backward, 0 means the nearest position to the steering wheel, 100% means the furthest position from the steering wheel	Since v5.0
Seat Vertical Position	verticalPosition	0-100%	Get/Set/Notification	Adjust seat height (up or down) in case there is only one actuator for seat height, 0 means the lowest position, 100% means the highest position	Since v5.0

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

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Seat Back Tilt Angle	backTiltAngle	0-100%	Get/Set/Notification	Backrest recline, 0 means the angle that back top is nearest to the the steering wheel, 100% means the angle that back top is furthest from the steering wheel	Since v5.0
Head Support Horizontal Position	headSupportHorizontalPosition	0-100%	Get/Set/Notification	Adjust head support forward/backward, 0 means the nearest position to the front, 100% means the furthest position from the front	Since v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Head Support Vertical Position	headSupportVerticalPosition	0-100%	Get/Set/Notification	Adjust head support height (up or down), 0 means the lowest position, 100% means the highest position	Since v5.0
Seat Massaging Enabled	massageEnabled	true, false	Get/Set/Notification	Indicates whether massage is enabled for a seat	Since v5.0
Massage Mode	massageMode	MassageModeData struct	Get/Set/Notification	List of massage mode of each zone	Since v5.0
Massage Cushion Firmness	massageCushionFirmness	MassageCushionFirmness struct	Get/Set/Notification	List of firmness of each massage cushion	Since v5.0
Seat memory	memory	SeatMemoryAction struct	Get/Set/Notification	Seat memory	Since v5.0

AUDIO

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Audio Volume	volume	0%-100%	Get/Set/Notification	The audio source volume level	Since SDL v5.0
Audio Source	source	PrimaryAudioSource enum	Get/Set/Notification	Defines one of the available audio sources	Since SDL v5.0
Keep Context	keepContext	true, false	Set only	Controls whether the HMI will keep the current application context or switch to the default media UI/APP associated with the audio source	Since SDL v5.0

CONTROL ITEM	RPC ITEM NAME	VALUE RANGE	TYPE	COMMENTS	RPC VERSION CHANGES
Equalizer Settings	equalizerSettings	EqualizerSettings struct	Get/Set/Notification	Defines the list of supported channels (band) and their current/desired settings on HMI	Since SDL v5.0

LIGHT

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

HMI SETTINGS

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

Remote Control Button Presses

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

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

RC MODULE	CONTROL BUTTON
	SHUFFLE
	REPEAT

Integration

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

Multiple Modules (RPC v6.0+)

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

Getting Remote Control Module Information

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

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

```
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.R
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        RemoteControlCapabilities remoteControlCapabilities =
(RemoteControlCapabilities) capability;
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        <# Handle Error #>
    }
});
```

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

With the saved remote control capabilities struct you can build a UI to display modules to the user by getting the location of the module and the area that it services. This will map to the grid you receive in **Setting the User's Seat** below.

NOTE

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

```
// Get the first climate module's information
ClimateControlCapabilities firstClimateModule = <#Remote Control
Capabilities#>.getClimateControlCapabilities().get(0);

String climateModuleId = firstClimateModule.getModuleInfo().getModuleId();
Grid climateModuleLocation =
firstClimateModule.getModuleInfo().getModuleLocation();
```

Setting The User's Seat (RPC v6.0+)

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

An array of seats can be found in the `seatLocationCapability` 's `seat` array. Each `SeatLocation` object within the `seats` array will have a `grid` parameter. The `grid` will tell you the seat placement of that particular seat. This information is useful for creating a seat location map from which users can select their seat.

```
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(Syste
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        SeatLocationCapability seatLocationCapability = (SeatLocationCapability)
capability;
        if (seatLocationCapability.getSeatLocations() != null &&
seatLocationCapability.getSeatLocations().size() > 0){
            List<SeatLocation> seats = seatLocationCapability.getSeatLocations();

            <#Save seat location capabilities#>
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        <#Handle Error#>
    }
});
```

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

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

UPDATING THE USER'S SEAT LOCATION

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

```
SetGlobalProperties seatLocation = new SetGlobalProperties();
seatLocation.setUserLocation(<#Selected Seat#>;);
seatLocation.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Seat location updated#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        <#Handle Error#>
    }
});
sdlManager.sendRPC(seatLocation);
```

Getting Module Data

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

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

SUBSCRIBING TO MODULE DATA

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

NOTE

Subscribing to the `OnInteriorVehicleData` notification must be done before sending the `GetInteriorVehicleData` request.

```
sdlManager.addOnRPCNotificationListener(FunctionID.ON_INTERIOR_VEHICLE_D
 new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnInteriorVehicleData onInteriorVehicleData = (OnInteriorVehicleData)
notification;
        if (onInteriorVehicleData != null){
            // NOTE: If you subscribe to multiple modules, all the data will be sent
here. You will have to
            // split it out based on
`onInteriorVehicleData.getModuleData().getModuleType()` yourself.
            <#Code#>
        }
    }
});
```

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

RPC < v6.0

```
GetInteriorVehicleData getInteriorVehicleData = new
GetInteriorVehicleData(ModuleType.RADIO);
getInteriorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // This can now be used to retrieve data
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(getInteriorVehicleData);
```

RPC v6.0+

```
GetInteriorVehicleData getInteriorVehicleData = new
GetInteriorVehicleData(ModuleType.RADIO);
getInteriorVehicleData.setModuleId(<#ModuleID#>);
getInteriorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // This can now be used to retrieve data
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(getInteriorVehicleData);
```

GETTING ONE-TIME DATA

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

RPC < v6.0

```
GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData(ModuleType.RADIO);
interiorVehicleData.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // This can now be used to retrieve data
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(interiorVehicleData);
```

RPC 6.0+

```
GetInteriorVehicleData interiorVehicleData = new
GetInteriorVehicleData(ModuleType.RADIO);
interiorVehicleData.setModuleId("<#ModuleID#>");
interiorVehicleData.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        // This can now be used to retrieve data
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(interiorVehicleData);
```

Setting Module Data

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

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

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

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

NOTE

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

```
GetInteriorVehicleDataConsent getInteriorVehicleDataConsent = new
GetInteriorVehicleDataConsent(<#ModuleType#>,<#ModuleIDs#>,);
getInteriorVehicleDataConsent.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        GetInteriorVehicleDataConsentResponse
getInteriorVehicleDataConsentResponse =
(GetInteriorVehicleDataConsentResponse) response;
        List<Boolean> allowed =
getInteriorVehicleDataConsentResponse.getAllowances();
            <#Allowed is an array of true or false values#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(getInteriorVehicleDataConsent);
```

CONTROLLING A MODULE

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

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

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

RPC < v6.0

```
Temperature temp = new Temperature(TemperatureUnit.FAHRENHEIT, 74.1f);

ClimateControlData climateControlData = new ClimateControlData();
climateControlData.setAcEnable(true);
climateControlData.setAcMaxEnable(true);
climateControlData.setAutoModeEnable(false);
climateControlData.setCirculateAirEnable(true);
climateControlData.setCurrentTemperature(temp);
climateControlData.setDefrostZone(DefrostZone.FRONT);
climateControlData.setDualModeEnable(true);
climateControlData.setFanSpeed(2);
climateControlData.setVentilationMode(VentilationMode.BOTH);
climateControlData.setDesiredTemperature(temp);

ModuleData moduleData = new ModuleData(ModuleType.CLIMATE);
moduleData.setClimateControlData(climateControlData);

SetInteriorVehicleData setInteriorVehicleData = new
SetInteriorVehicleData(moduleData);
setInteriorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(setInteriorVehicleData);
```

RPC 6.0+

```
Temperature temp = new Temperature(TemperatureUnit.FAHRENHEIT, 74.1f);

ClimateControlData climateControlData = new ClimateControlData();
climateControlData.setAcEnable(true);
climateControlData.setAcMaxEnable(true);
climateControlData.setAutoModeEnable(false);
climateControlData.setCirculateAirEnable(true);
climateControlData.setCurrentTemperature(temp);
climateControlData.setDefrostZone(DefrostZone.FRONT);
climateControlData.setDualModeEnable(true);
climateControlData.setFanSpeed(2);
climateControlData.setVentilationMode(VentilationMode.BOTH);
climateControlData.setDesiredTemperature(temp);

ModuleData moduleData = new ModuleData(ModuleType.CLIMATE);
moduleData.setModuleId("<#ModuleID#>");
moduleData.setClimateControlData(climateControlData);

SetInteriorVehicleData setInteriorVehicleData = new
SetInteriorVehicleData(moduleData);
setInteriorVehicleData.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(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 to simulate.

RPC < 6.0

```
ButtonPress buttonPress = new ButtonPress(ModuleType.RADIO,
ButtonName.EJECT, ButtonPressMode.SHORT);
buttonPress.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(buttonPress);
```

RPC 6.0+

```
ButtonPress buttonPress = new ButtonPress(ModuleType.RADIO,
ButtonName.EJECT, ButtonPressMode.SHORT);
buttonPress.setModuleId("<#ModuleID#>");
buttonPress.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Code#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(buttonPress);
```

RELEASING THE MODULE (RPC V6.0+)

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

```
ReleaseInteriorVehicleDataModule releaseInteriorVehicleDataModule = new
ReleaseInteriorVehicleDataModule(<#ModuleType#>);
releaseInteriorVehicleDataModule.setModuleId(<#ModuleID#>);
releaseInteriorVehicleDataModule.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Module Was Released#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle Error#>
    }
});
sdlManager.sendRPC(releaseInteriorVehicleDataModule);
```

Creating an App Service (RPC v5.1+)

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

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

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

An SDL app can also subscribe to a published app service. Once subscribed, the app will be sent the new data when the app service publisher updates its data. To find out more about how to subscribe to an app service check out the Using App Services 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 for each of the different service types) if desired.

Publishing an App Service

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

1. Creating an App Service Manifest

The first step to publishing an app service is to create an `AppServiceManifest` 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.

```
AppServiceManifest manifest = new
AppServiceManifest(AppServiceType.MEDIA.toString());
manifest.setServiceName("My Media App"); // Must be unique across app
services.
manifest.setServiceIcon(new Image("Service Icon Name", ImageType.DYNAMIC));
// Previously uploaded service icon. This could be the same as your app icon.
manifest.setAllowAppConsumers(true); // Whether or not other apps can view
your data in addition to the head unit. If set to `false` only the head unit will have
access to this data.
manifest.setRpcSpecVersion(new SdlMsgVersion(5,0)); // An *optional*
parameter that limits the RPC spec versions you can understand to the provided
version *or below*.
manifest.setHandledRpcs(List<FunctionID>); // 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.setMediaServiceManifest(<#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.

```
MediaServiceManifest mediaManifest = new MediaServiceManifest();
manifest.setMediaServiceManifest(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).

```
NavigationServiceManifest navigationManifest = new
NavigationServiceManifest();
navigationManifest.setAcceptsWayPoints(true);
manifest.setNavigationServiceManifest(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 `WeatherServiceData` .

```
WeatherServiceManifest weatherManifest = new WeatherServiceManifest();
weatherManifest.setCurrentForecastSupported(true);
weatherManifest.setMaxMultidayForecastAmount(10);
weatherManifest.setMaxHourlyForecastAmount(24);
weatherManifest.setMaxMinutelyForecastAmount(60);
weatherManifest.setWeatherForLocationSupported(true);
manifest.setWeatherServiceManifest(weatherManifest);
```

2. Publish Your Service

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

```
PublishAppService publishServiceRequest = new PublishAppService();
publishServiceRequest.setAppServiceManifest(manifest);
publishServiceRequest.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Use the response#>
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Error Handling#>
    }
});
sdlManager.sendRPC(publishServiceRequest);
```

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 `AppServiceRecord` 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 `PublishAppServiceResponse`, 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 `SystemCapabilityType.APP_SERVICES` using `GetSystemCapability` and `OnSystemCapabilityUpdated`.

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 .

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 `MediaServiceData`, `NavigationServiceData` or `WeatherServiceData` object with your service's data. Then, add that service-specific data object to an `AppServiceData` object. Finally, create an `OnAppServiceData` notification, append your `AppServiceData` object, and send it.

MEDIA SERVICE DATA

```
MediaServiceData mediaData = new MediaServiceData();
mediaData.setMediaTitle("Some media title");
mediaData.setMediaArtist("Some media artist");
mediaData.setMediaAlbum("Some album");
mediaData.setMediaImage(new Image("Some image", ImageType.DYNAMIC));
mediaData.setPlaylistName("Some playlist");
mediaData.setIsExplicit(true);
mediaData.setTrackPlaybackProgress(45);
mediaData.setTrackPlaybackDuration(90);
mediaData.setQueuePlaybackProgress(45);
mediaData.setQueuePlaybackDuration(150);
mediaData.setQueueCurrentTrackNumber(2);
mediaData.setQueueTotalTrackCount(3);

AppServiceData appData = new AppServiceData();
appData.setServiceID(myServiceId);
appData.setServiceType(AppServiceType.MEDIA.toString());
appData.setMediaServiceData(mediaData);

OnAppServiceData onAppData = new OnAppServiceData();
onAppData.setServiceData(appData);

sdlManager.sendRPC(onAppData);
```

NAVIGATION SERVICE DATA

```
final SdlArtwork navInstructionArt = new SdlArtwork("turn",
FileType.GRAPHIC_PNG, R.drawable.turn, true);

sdlManager.getFileManager().uploadFile(navInstructionArt, new
CompletionListener() { // We have to send the image to the system before it's
used in the app service.
    @Override
    public void onComplete(boolean success) {
        if (success){
            Coordinate coordinate = new Coordinate(42f,43f);

            LocationDetails locationDetails = new LocationDetails();
            locationDetails.setCoordinate(coordinate);

            NavigationInstruction navigationInstruction = new
NavigationInstruction(locationDetails, NavigationAction.TURN);
            navigationInstruction.setImage(navInstructionArt.getImageRPC());

            DateTime dateTime = new DateTime();
            dateTime.setHour(2);
            dateTime.setMinute(3);
            dateTime.setSecond(4);

            NavigationServiceData navigationData = new
NavigationServiceData(dateTime);

navigationData.setInstructions(Collections.singletonList(navigationInstruction));

            AppServiceData appData = new AppServiceData();
            appData.setServiceID(myServiceId);
            appData.setServiceType(AppServiceType.NAVIGATION.toString());
            appData.setNavigationServiceData(navigationData);

            OnAppServiceData onAppData = new OnAppServiceData();
            onAppData.setServiceData(appData);

            sdlManager.sendRPC(onAppData);
        }
    }
});
```

WEATHER SERVICE DATA

```
final SdlArtwork weatherImage = new SdlArtwork("sun", FileType.GRAPHIC_PNG,
R.drawable.sun, true);

sdlManager.getFileManager().uploadFile(weatherImage, new
CompletionListener() { // We have to send the image to the system before it's
used in the app service.
    @Override
    public void onComplete(boolean success) {
        if (success) {

            WeatherData weatherData = new WeatherData();
            weatherData.setWeatherIcon(weatherImage.getImageRPC());

            Coordinate coordinate = new Coordinate(42f, 43f);

            LocationDetails locationDetails = new LocationDetails();
            locationDetails.setCoordinate(coordinate);

            WeatherServiceData weatherServiceData = new
WeatherServiceData(locationDetails);

            AppServiceData appData = new AppServiceData();
            appData.setServiceID(myServiceId);
            appData.setServiceType(AppServiceType.WEATHER.toString());
            appData.setWeatherServiceData(weatherServiceData);

            OnAppServiceData onAppData = new OnAppServiceData();
            onAppData.setServiceData(appData);

            sdlManager.sendRPC(onAppData);
        }
    }
});
```

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 setup listeners for 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 setup a listener for a `GetAppServiceDataRequest`.

```
// Get App Service Data Request Listener
sdlManager.addOnRPCRequestListener(FunctionID.GET_APP_SERVICE_DATA,
new OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        <#Handle Request#>
    }
});
```

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).

```
// Get App Service Data Request Listener
sdlManager.addOnRPCRequestListener(FunctionID.GET_APP_SERVICE_DATA,
new OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        GetAppServiceData getAppServiceData = (GetAppServiceData) request;

        GetAppServiceDataResponse response = new
GetAppServiceDataResponse();
        response.setSuccess(true);
        response.setCorrelationID(getAppServiceData.getCorrelationID());
        response.setResultCode(Result.SUCCESS);
        response.setInfo("<#Use to provide more information about an error#>");
        response.setServiceData(<#Your App Service Data#>);

        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 listeners to be aware that they have arrived, and you will then need to respond to those requests.

```
AppServiceManifest manifest = new
AppServiceManifest(AppServiceType.MEDIA.toString());
...
manifest.setHandledRpcs(Collections.singletonList(FunctionID.BUTTON_PRESS.g
```

```
sdlManager.addOnRPCRequestListener(FunctionID.BUTTON_PRESS, new
OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        ButtonPress buttonPress = (ButtonPress) request;

        ButtonPressResponse response = new ButtonPressResponse();
        response.setSuccess(true);
        response.setResultCode(Result.SUCCESS);
        response.setCorrelationID(buttonPress.getCorrelationID());
        response.setInfo("<#Use to provide more information about an error#>");
        sdlManager.sendRPC(response);
    }
});
```

6. Service Actions

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

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

```
// Perform App Services Interaction Request Listener
sdlManager.addOnRPCRequestListener(FunctionID.PERFORM_APP_SERVICES_IN
 new OnRPCRequestListener() {
    @Override
    public void onRequest(RPCRequest request) {
        PerformAppServiceInteraction performAppServiceInteraction =
(PerformAppServiceInteraction) request;

        // If you have multiple services, this will let you know which of your services
is being addressed
        serviceID = performAppServiceInteraction.getServiceID();

        // The URI sent by the consumer. This must be something you understand
        String serviceURI = performAppServiceInteraction.getServiceUri();

        // A result you want to send to the consumer app.
        PerformAppServiceInteractionResponse response = new
PerformAppServiceInteractionResponse();
        response.setServiceSpecificResult("Some Result");

response.setCorrelationID(performAppServiceInteraction.getCorrelationID());
        response.setInfo("<#Use to provide more information about an error#>");
        response.setSuccess(true);
        response.setResultCode(Result.SUCCESS);
        sdlManager.sendRPC(response);
    }
});
```

Updating Your Published App Service

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

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

```
AppServiceManifest manifest = new
AppServiceManifest(AppServiceType.WEATHER.toString());
manifest.setWeatherServiceManifest("<#Updated weather service manifest>");

PublishAppService publishServiceRequest = new PublishAppService(manifest);
sdlManager.sendRPC(publishServiceRequest);
```

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

```
UnpublishAppService unpublishAppService = new UnpublishAppService("<#The
serviceID of the service to unpublish>");
sdlManager.sendRPC(unpublishAppService);
```

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 `SystemCapabilityManager` to get the information. Because this information is initially available asynchronously, we have to attach an `OnSystemCapabilityListener` to the `getCapability` request.

JAVA

```
// Grab the capability once
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.A
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        AppServicesCapabilities servicesCapabilities = (AppServicesCapabilities)
capability;
    }

    @Override
    public void onError(String info) {
        <# Handle Error #>
    }
});

...

// Subscribe to updates
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(Syste
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        AppServicesCapabilities servicesCapabilities = (AppServicesCapabilities)
capability;
    }

    @Override
    public void onError(String info) {
        <# Handle Error #>
    }
});
```

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.

JAVA

```
// This array contains all currently available app services on the system
List<AppServiceCapability> appServices = servicesCapabilities.getAppServices();

if (appServices!= null && appServices.size() > 0) {
   for (AppServiceCapability anAppServiceCapability : appServices) {
      // This will tell you why a service is in the list of updates
      ServiceUpdateReason updateReason =
anAppServiceCapability.getUpdateReason();

      // 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)
      AppServiceRecord serviceRecord =
anAppServiceCapability.getUpdatedAppServiceRecord();
   }
}
```

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."

JAVA

```
// Get service data once
GetAppServiceData getAppServiceData = new
GetAppServiceData(AppServiceType.MEDIA.toString());

// Subscribe to future updates if you want them
getAppServiceData.setSubscribe(true);

getAppServiceData.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response != null){
            GetAppServiceDataResponse serviceResponse =
(GetAppServiceDataResponse) response;
            MediaServiceData mediaServiceData =
serviceResponse.getServiceData().getMediaServiceData();
        }
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <# Handle Error #>
    }
});
sdlManager.sendRPC(getAppServiceData);

...

// Unsubscribe from updates
GetAppServiceData unsubscribeServiceData = new
GetAppServiceData(AppServiceType.MEDIA.toString());
unsubscribeServiceData.setSubscribe(false);
sdlManager.sendRPC(unsubscribeServiceData);
```

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.

JAVA

```
ButtonPress buttonPress = new ButtonPress();
buttonPress.setButtonPressMode(ButtonPressMode.SHORT);
buttonPress.setButtonName(ButtonName.OK);
buttonPress.setModuleType(ModuleType.AUDIO);
buttonPress.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Use the response#>
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle the Error#>
    }
});
sdlManager.sendRPC(buttonPress);
```

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).

JAVA

```
PerformAppServiceInteraction performAppServiceInteraction = new
PerformAppServiceInteraction("sdlexample://x-callback-url/showText?x-
source=MyApp&text=My%20Custom%20String","<#Previously Retrieved
ServiceID#>","<#Your App Id#>");
performAppServiceInteraction.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        <#Use the response#>
    }
    @Override
    public void onError(int correlationId, Result resultCode, String info){
        <#Handle the Error#>
    }
});
sdlManager.sendRPC(performAppServiceInteraction);
```

5. Getting a File from a Service Provider

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

```
AppServiceData appServiceData = <#Get the App Service Data#>;
WeatherServiceData weatherServiceData =
appServiceData.getWeatherServiceData();
if (weatherServiceData == null || weatherServiceData.getCurrentForecast() == null
|| weatherServiceData.getCurrentForecast().getWeatherIcon() == null) {
    // The image doesn't exist, exit early
    return;
}
String currentForecastImageName =
weatherServiceData.getCurrentForecast().getWeatherIcon().getValue();

GetFile getFile = new GetFile(currentForecastImageName);
getFile.setAppServiceId(<#Service ID>);
getFile.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        GetFileResponse getFileResponse = (GetFileResponse) response;
        byte[] fileData = getFileResponse.getBulkData();
        SdlArtwork sdlArtwork = new SdlArtwork(fileName, FileType.GRAPHIC_PNG,
fileData, false);
        // Use the sdlArtwork
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        // Something went wrong, examine the resultCode and info
    }
});
sdlManager.sendRPC(getFile);
```

Calling a Phone Number

The `DialNumber` 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

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

Checking if Dial Number is Available

`DialNumber` 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.isPhoneCallAvailable()` property after the manager has been started successfully.

```
HMICapabilities hmiCapabilities =
(HMICapabilities)sdlManager.getSystemCapabilityManager().getCapability(Systen

if (hmiCapabilities.isPhoneCallAvailable()) {
    // DialNumber supported
} else {
    // DialNumber is not supported
}
```

Sending a DialNumber Request

NOTE

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

```
DialNumber dialNumber = new DialNumber();
dialNumber.setNumber("1238675309");
dialNumber.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        Result result = response.getResultCode();
        if(result.equals(Result.SUCCESS)){
            // `DialNumber` was successfully sent, and a phone call was initiated by
the user.
        }else if(result.equals(Result.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.
        }else if(result.equals(Result.DISALLOWED)){
            // Your app does not have permission to use DialNumber.
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(dialNumber);
```

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 SendLocation

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 SendLocation

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.4.4+, you can use the `SystemCapabilityManager` to check the navigation capability returned by Core as shown in the code sample below.

```
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.N
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        NavigationCapability navCapability = (NavigationCapability) capability;
        Boolean isNavigationSupported = navCapability.getSendLocationEnabled();
    }

    @Override
    public void onError(String info) {
        HMICapabilities hmiCapabilities = (HMICapabilities)
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.H

        Boolean isNavigationSupported = hmiCapabilities.isNavigationAvailable();
    }
});
```

Using Send Location

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

```
SendLocation sendLocation = new SendLocation();
sendLocation.setLatitudeDegrees(42.877737);
sendLocation.setLongitudeDegrees(-97.380967);
sendLocation.setLocationName("The Center");
sendLocation.setLocationDescription("Center of the United States");

// Create Address
OasisAddress address = new OasisAddress();
address.setSubThoroughfare("900");
address.setThoroughfare("Whiting Dr");
address.setLocality("Yankton");
address.setAdministrativeArea("SD");
address.setPostalCode("57078");
address.setCountryCode("US-SD");
address.setCountryName("United States");

sendLocation.setAddress(address);

// Monitor response
sendLocation.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        Result result = response.getResultCode();
        if(result.equals(Result.SUCCESS)){
            // SendLocation was successfully sent.
        }else if(result.equals(Result.INVALID_DATA)){
            // The request you sent contains invalid data and was rejected.
        }else if(result.equals(Result.DISALLOWED)){
            // Your app does not have permission to use SendLocation.
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});

sdlManager.sendRPC(sendLocation);
```

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.

Getting the Navigation Destination

The GetWayPoints and SubscribeWayPoints RPCs are designed to allow you to get the navigation destination(s) from the active navigation app if the user is navigating.

Checking If Your App Has Permission to Use GetWayPoints

The GetWayPoints and SubscribeWayPoints RPCs are 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 GetWaypoints

Since there is a possibility that some head units will not support getting the navigation destination, 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 GetWayPoints.

You can use the SystemCapabilityManager to check the navigation capability returned by Core as shown in the code sample below.

```
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.N
 new OnSystemCapabilityListener() {
    @Override
    public void onCapabilityRetrieved(Object capability) {
        NavigationCapability navCapability = (NavigationCapability) capability;
        boolean isNavigationSupported = navCapability != null &&
navCapability.getWayPointsEnabled();
    }

    @Override
    public void onError(String info) {
        HMICapabilities hmiCapabilities = (HMICapabilities)
sdlManager.getSystemCapabilityManager().getCapability(SystemCapabilityType.H

        boolean isNavigationSupported = hmiCapabilities.isNavigationAvailable();
    }
});
```

Subscribing to WayPoints

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

JAVA

```
// Create this method to receive the subscription callback
sdlManager.addOnRPCNotificationListener(FunctionID.ON_WAY_POINT_CHANGE,
 new OnRPCNotificationListener() {
    @Override
    public void onNotified(RPCNotification notification) {
        OnWayPointChange onWayPointChangeNotification = (OnWayPointChange)
notification;
        //<#Use the waypoint data#>
    }
});

// After SDL has started your connection, at whatever point you want to
subscribe, send the subscribe RPC
SubscribeWayPoints subscribeWayPoints = new SubscribeWayPoints();
subscribeWayPoints.setOnRPCResponseListener(new OnRPCResponseListener()
{
    @Override
    public void onResponse(int correlationId, RPCResponse rpcResponse) {
        if (rpcResponse.getSuccess()){
            // You are now subscribed!
        } else {
            // Handle the errors
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        // Handle the errors
    }
});
sdlManager.sendRPC(subscribeWayPoints);
```

Unsubscribing from Waypoints

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

JAVA

```
UnsubscribeWayPoints unsubscribeWayPoints = new UnsubscribeWayPoints();
unsubscribeWayPoints.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse rpcResponse) {
        if (rpcResponse.getSuccess()){
            // You are now unsubscribed!
        } else {
            // Handle the errors
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        // Handle the errors
    }
});
sdlManager.sendRPC(unsubscribeWayPoints);
```

One-Time Waypoints Request

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

JAVA

```
GetWayPoints getWayPoints = new GetWayPoints();
getWayPoints.setOnRPCResponseListener(new OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse rpcResponse) {
        if (rpcResponse.getSuccess()){
            GetWayPointsResponse getWayPointsResponse =
(GetWayPointsResponse) rpcResponse;
            <#Use the waypoint information#>
        } else {
            // Handle the errors
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info) {
        // Handle the errors
    }
});
sdlManager.sendRPC(getWayPoints);
```

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 `FileManager` 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 `SdlAr` `twork` object. Both `SdlFile`s and `SdlArtwork`s can be created with using `filePath`, or `byte[]`.

```
byte[] mp3Data = Get the file data;
SdlFile audioFile = new SdlFile("File Name", FileType.AUDIO_MP3, mp3Data, true);
sdlManager.sendRPC(audioFile);
```

Batching File Uploads

If you want to upload a group of files, you can use the `FileManager` batch upload methods. Once all of the uploads have completed you will be notified if any of the uploads failed.

```
sdlManager.getFileManager().uploadFiles(sdlFileList, new
MultipleFileCompletionListener() {
    @Override
    public void onComplete(Map<String, String> errors) {

    }
});
```

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:

```
Boolean isPersistent = file.isPersistent();
```

NOTE

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

Checking the Amount of File Storage Left

To find the amount of file storage left for your app on the head unit, use the `ListFiles` RPC.

```
int bytesAvailable = sdlManager.getFileManager().getBytesAvailable();
```

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 `FileManager`'s `remoteFileNames` property.

```
Boolean fileIsOnHeadUnit =
sdlManager.getFileManager().getRemoteFileNames().contains("Name Uploaded
As")
```

Deleting Stored Files

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

```
sdlManager.getFileManager().deleteRemoteFileWithName("Name Uploaded As",
new CompletionListener() {
    @Override
    public void onComplete(boolean success) {

    }
});
```

Batch Deleting Files

```
sdlManager.getFileManager().deleteRemoteFilesWithNames(remoteFiles, new
MultipleFileCompletionListener() {
    @Override
    public void onComplete(Map<String, String> errors) {

    }
});
```

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, check the `getCapability()` method of a valid `SystemCapabilityManager` obtained from `sdlManager.getSystemCapabilityManager()` to find out the display capabilities of the head unit.

```
List<ImageField> imageFields =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().ge

boolean imagesSuported = (imageFields.size() > 0);
```

Uploading an Image Using the File Manager

The `FileManager` uploads files and keeps track of all the uploaded files names during a session. To send data with the `FileManager`, you need to create either a `SdlFile` or `SdlArtwork` object. Both `SdlFile`s and `SdlArtwork`s can be created with using `filePath`, or `byte[]`.

```
SdlArtwork artwork = new SdlArtwork("image_name", FileType.GRAPHIC_PNG,
<image byte[]>, false);
sdlManager.getFileManager().uploadFile(artwork, new CompletionListener() {
    @Override
    public void onComplete(boolean success) {
        if (success){
            <#Image Upload Successful#>
        }
    }
});
```

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.4.8 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:

```
CloudAppProperties cloudAppProperties = new CloudAppProperties("<appId>");
cloudAppProperties.setAuthToken("<auth token>");
SetCloudAppProperties setCloudAppProperties = new
SetCloudAppProperties(cloudAppProperties);
setCloudAppProperties.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()) {
            Log.i("SdlService", "Request was successful.");
        } else {
            Log.i("SdlService", "Request was rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(setCloudAppProperties);
```

Getting Cloud App Properties

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

```
GetCloudAppProperties getCloudAppProperties = new GetCloudAppProperties("
<appId>");
getCloudAppProperties.setOnRPCResponseListener(new
OnRPCResponseListener() {
    @Override
    public void onResponse(int correlationId, RPCResponse response) {
        if (response.getSuccess()) {
            Log.i("SdlService", "Request was successful.");
            GetCloudAppPropertiesResponse getCloudAppPropertiesResponse =
(GetCloudAppPropertiesResponse) response;
            CloudAppProperties cloudAppProperties =
getCloudAppPropertiesResponse.getCloudAppProperties();
            // Use cloudAppProperties
        } else {
            Log.i("SdlService", "Request was rejected.");
        }
    }

    @Override
    public void onError(int correlationId, Result resultCode, String info){
        Log.e(TAG, "onError: "+ resultCode+ " | Info: "+ info );
    }
});
sdlManager.sendRPC(getCloudAppProperties);
```

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:

```
String authToken = sdlManager.getAuthToken();
```

Getting CloudAppVehicleID (Optional)

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

Encryption

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

When Encryption is Needed

OEM Required Encrypted RPCs

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

Optional Encryption

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

NOTE

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

Creating the Encryption Configuration

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

```
List<Class<? extends SdlSecurityBase>> secList = new ArrayList<>();
secList.add(OEMSdlSecurity.class);
builder.setSdlSecurity(secList, <# Optional serviceEncryptionListener>);
```

Getting the Encryption Status

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

```
ServiceEncryptionListener serviceEncryptionListener = new
ServiceEncryptionListener() {
    @Override
    public void onEncryptionServiceUpdated(@NonNull SessionType serviceType,
boolean isServiceEncrypted, @Nullable String error) {
        if (isServiceEncrypted) {
            // Encryption manager can encrypt
        }
    }
};
```

Setting Optional Encryption

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

```
sdlManager.startRPCEncryption();
```

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

```
GetVehicleData getVehicleData = new GetVehicleData();
getVehicleData.setGps(true);
getVehicleData.setPayloadProtected(true);

sdlManager.sendRPC(getVehicleData);
```

Configuring SDL Logging

SDL Java Suite has a built-in logging framework that is designed to make debugging easier. It provides many of the features common to other 3rd party logging frameworks for java 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 `System.out.print`. This will consolidate all SDL related logs into a common format and to a common destination.

Enabling the DebugTool

To make sure that log messages are displayed, you should enable the SDL Debug Tool:

```
DebugTool.enableDebugTool();
```

If you don't want the messages to be logged, you can disable the Debug Tool anytime:

```
DebugTool.disableDebugTool();
```

NOTE

If you use SDL Debug Tool to log messages without enabling the DebugTool nothing wrong will happen. It will simply not display the log messages. This gives the develop control on whether the logs should be displayed or not.

Logging messages

The SDL debug tool can be used to log messages with different log levels. The log level defines how serious the log message is. This table summarizes when to use each log level:

LOG LEVEL	WHEN TO USE
Info	Use this to post useful information to the log
Warning	Use this when you suspect something shady is going on
Error	Use this when bad stuff happens

To log an info message:

```
DebugTool.logInfo("info message goes here");
```

To log a warning message:

```
DebugTool.logWarning("warning message goes here");
```

To log an error message:

```
DebugTool.logError("error message goes here");
```

If you want to log error message with exception, you can add the exception as a second parameter to the `logError` method:

```
DebugTool.logError("error message goes here", new SdlException("Sdl
connection failed", SdlExceptionCause.SDL_CONNECTION_FAILED));
```

Upgrading to 4.9

Overview

This guide is to help developers get setup with the SDL Java library version 4.9. It is assumed that the developer is already updated to at least version 4.7 or 4.8 of the library.

The full release notes are published here.

The main differences between the previous release and this are mainly additive, including 3 new managers which we will describe briefly. Additionally, we have fixed an issue where symlinks were not working on Windows machines by creating a gradle task that builds them for you. Additionally, we have added the ability to pass a buffer to the AudioStreamManager to play raw data.

Voice Command Manager

The voice command manager is accessed via the `ScreenManager`. It allows for an easy way to create global voice commands for your application. These are not supposed to be a replacement for menu voice commands, but rather an easy way to trigger main events in your application, similar to something you might use a `SoftButton` for. These commands, once sent, will be available on the system as voice commands for the duration of the session.

An example is as follows:

```
List<String> list1 = Collections.singletonList("Command One");
List<String> list2 = Collections.singletonList("Command two");

VoiceCommand voiceCommand1 = new VoiceCommand(list1, new
VoiceCommandSelectionListener() {
    @Override
    public void onVoiceCommandSelected() {
        Log.i(TAG, "Voice Command 1 triggered");
    }
});

VoiceCommand voiceCommand2 = new VoiceCommand(list2, new
VoiceCommandSelectionListener() {
    @Override
    public void onVoiceCommandSelected() {
        Log.i(TAG, "Voice Command 2 triggered");
    }
});

sdlManager.getScreenManager().setVoiceCommands(Arrays.asList(voiceComma
```

Menu Manager

Menus have now become simpler with the `MenuManager`, which is accessed via the `ScreenManager`. The cells, called `MenuCell`'s contain 2 constructors. One is for a cell itself, and the other is a cell that contains a sub-menu. Note that currently SmartDeviceLink (SDL) only supports sub-menus to the depth of 1.

`MenuCell`s contain a `MenuSelectionListener` which informs you that the cell has been triggered, so that you might perform an action based on the cell selected. Note that you can add images and voice commands to menu cells.

NOTE

When submitting a list of Menu cells, or adding a list of sub cells to a menu cell, the order in which the cells will appear from top to bottom will be the order in which they are in the list.

Example use:

```
// SUB MENU CELLS FOR MAIN MENU CELL 2

// Sub cells are just normal cells
MenuCell subCell1 = new MenuCell("SubCell 1",null, null, new
MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        Log.i(TAG, "Sub cell 1 triggered. Source: "+ trigger.toString());
    }
});

MenuCell subCell2 = new MenuCell("SubCell 2",null, null, new
MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        Log.i(TAG, "Sub cell 2 triggered. Source: "+ trigger.toString());
    }
});

// THE MAIN MENU CELLS

// normal cell
MenuCell mainCell1 = new MenuCell("Test Cell 1 (speak)", null, null, new
MenuSelectionListener() {
    @Override
    public void onTriggered(TriggerSource trigger) {
        Log.i(TAG, "Test cell 1 triggered. Source: "+ trigger.toString());
    }
});

// sub menu parent cell
MenuCell mainCell2 = new MenuCell("Test Cell 3 (sub menu)", null,
Arrays.asList(subCell1,subCell2));

// Send the entire menu off to be created
sdlManager.getScreenManager().setMenu(Arrays.asList(mainCell1, mainCell2));
```

Choice Set Manager

Previously it required a lot of code to use `PerformInteraction`s with SDL. To alleviate some of this pain, we have introduced the Choice Set Manager which is accessible via the `ScreenManager`. Because the Choice Set Manager covers so many items, we will do a brief overview here. You may continue to the Popup Menus and Keyboards section for more detailed information.

There are 2 main use cases for using this manager, one is to display a choice set, and the other is to display a keyboard.

Choice Set

Displaying a choice set is achieved by creating some `ChoiceCell` s. If you know what your choices will be, we recommend using the `preloadChoices` method. This will ensure your `ChoiceSet` is ready to be displayed when you want to display it, and your user is not kept waiting. You can preload cells as follows:

```
// create some choice cells
ChoiceCell cell1 = new ChoiceCell("Item 1");
ChoiceCell cell2 = new ChoiceCell("Item 2");
ChoiceCell cell3 = new ChoiceCell("Item 3");

// create the array of choice cells
choiceCellList = Arrays.asList(cell1,cell2,cell3);

// pre-load the cells on the head unit
sdlManager.getScreenManager().preloadChoices(choiceCellList, null);
```

NOTE

You will want to reference this array of cells when presenting your choice set later (even if you add more cells). This is why we are setting this list to a variable for now.

Once you are ready to present the Choice Set, you can do so by:

```
ChoiceSet choiceSet = new ChoiceSet("Choose an Item from the list",
choiceCellList, new ChoiceSetSelectionListener() {
    @Override
    public void onChoiceSelected(ChoiceCell choiceCell, TriggerSource
triggerSource, int rowIndex) {
        // do something with the selection
    }

    @Override
    public void onError(String error) {
        Log.e(TAG, "There was an error showing the perform interaction: "+ error);
    }
});
sdlManager.getScreenManager().presentChoiceSet(choiceSet,
InteractionMode.MANUAL_ONLY);
```

Displaying A Keyboard

There is now also an easy way to display a keyboard, and listen for key events. You simply need a `KeyboardListener` object.

```
KeyboardListener keyboardListener = new KeyboardListener() {
    @Override
    public void onUserDidSubmitInput(String inputText, KeyboardEvent event) {

    }

    @Override
    public void onKeyboardDidAbortWithReason(KeyboardEvent event) {

    }

    @Override
    public void updateAutocompleteWithInput(String currentInputText,
KeyboardAutocompleteCompletionListener
keyboardAutocompleteCompletionListener) {

    }

    @Override
    public void updateCharacterSetWithInput(String currentInputText,
KeyboardCharacterSetCompletionListener
keyboardCharacterSetCompletionListener) {

    }

    @Override
    public void onKeyboardDidSendEvent(KeyboardEvent event, String
currentInputText) {

    }
};
```

You can note that two of the methods contain a `KeyboardAutocompleteCompletionListener` and a `KeyboardCharacterSetCompletionListener`. These listeners allow you to show auto completion text and to modify the available keys, respectively, on supported head units.

To actually display the keyboard, call:

```
sdlManager.getScreenManager().presentKeyboard("initialText", null,
keyboardListener);
```

The `null` parameter in this example is a `KeyboardProperties` object that you can optionally pass in to modify the keyboard for this request.