sdl

JavaScript Suite Guides

Document current as of 12/14/2022 12:26 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 Node.js v10.24.1 and above.

Install SDL SDK

You can find the most recent release of the SDL JavaScript Suite here. The project comes with prebuilt bundles of the library in the form of `SDL.min.js` files. There is a vanilla JavaScript distribution of the library as well as one for Node.js. They are located in the `lib/js/dist` and `lib/node/dist` directories respectively.

Vanilla JavaScript Setup

This build allows you to create apps that run on the browser. In order to have the built JS file be imported into your HTML, you'll need to run a simple web server that can serve that JS file. We will be using Node.js and npm for this task, but you can use any software that lets you serve HTML and JS to the browser.

In a new directory, save your `SDL.min.js` file there, and then run the following: `npm init`. Press Enter repeatedly through the prompts to set up some default npm configuration. Then, install the `express` package by running `npm install express --save`. Express is a popular Node.js package that allows easy setup of a server. Then, create a `index.js` file in the same directory, and save the following to it:

```
const express = require('express');
const app = express();
app.use(express.static(__dirname));
const PORT = process.env.PORT || 3000;
const server = app.listen(PORT, async function () {
    console.log('Server running on port', PORT);
});
```

This code will start up a server on port 3000 on localhost, and serve any files in the directory it is running in. Now that the server code is complete, create a new file named `index.html` that will contain the app logic and save it in the same directory. Make the contents of the HTML file the following:

```
<html>
    <head>
        <script src='./SDL.min.js'></script>
    </head>
    <body>
        <script>
            console.log(SDL);
        </script>
    </body>
</html>
```

Finally, run the server by running this in a Terminal window in the same directory: `node index.js`. The console should print `Server running on port 3000`. Go to your browser and enter `localhost:3000` in the address bar. If you open up your browser's console on the blank page that shows up you should see the SDL library version be printed and the imported SDL object. If so, then you have successfully set up SDL in your browser! If not, then make sure your SDL build name is correct, and that the HTML file, build file, and JS server file are all in the same directory. If you still have questions, ask them in the javascript-suite-help channel in the SDL Slack. You can sign up for our Slack here.

Node.js Setup

This build allows you to create apps that run through a server on your computer. You will need to have installed Node.js and npm are before beginning work on this app.

In a new directory, save your `SDL.min.js` file, then create a new file in the same directory named `index.js`. Make the contents of that file the following:

```
const SDL = require('./SDL.min.js');
console.log(SDL);
```

Finally, run the file by entering the following in a Terminal window in the same directory: `node index.js`. You should see the SDL library version and the imported SDL object be printed to the console. If so, then you have successfully set up the SDL library!

WebEngine App Setup

WebEngine apps use the vanilla JavaScript build, and are set up in a similar fashion to those JS apps where it will also run in the browser. Set up the `index.js` and `index.html` file like in the **Vanilla JavaScript Setup** section. The majority of the configuration for the app will now be separated into a `manifest.js` file and then imported into the `index.html` file. Create a `manifest.js` file like below and save it in the same directory as the other two files. The entrypoint's value should have the same name as your app's HTML file.

```
export default {
    "entrypoint": "./index.html",
    "applcon": "./app_icon.png",
    "appId": "hello-webengine",
    "appName": "Hello WebEngine",
    "category": "DEFAULT",
    "additionalCategories": [],
    "locales": {
        "de_DE": {
            "appName": "Hallo JS",
            "applcon": "./app_icon.png"
        }
    },
    "appVersion": "1.0.0",
    "minRpcVersion": "6.0",
    "minProtocolVersion": "5.0"
};
```

Note that the manifest file is using the import/export module syntax; in the HTML file it should be imported in as a module:

```
import sdl_manifest from './manifest.js';
```

In the transport configuration the parameters for `WebSocketClientConfig` will be empty. For WebEngine apps those connection details are expected as query parameters in the URL. See below for an example of what the URL is expected to be once the server is running. `sdl-host` is the location of the SDL Core WebSocket server. `sdl-port` is the port of that WebSocket server. `sdl-transport-role` refers to SDL Core's role, which is as a server (as opposed to a client).

```
http://localhost:3000/?sdl-host=HOST&sdl-transport-role=ws-server&sdl-port=PORT
```

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

The type of app you can make will depend on the build library you select. If you are using the Node.js build, your app can run as a WebSocket server or as a TCP client. If you are using the vanilla JavaScript build (a minified JS file not tied to any specific build system or server structure), your app can run as a WebSocket client. This guide will cover topics that apply to both Node.js and vanilla JS library builds.

Basic Configuration

In order to correctly connect to an SDL enabled head unit, developers need to create an `AppConfig` configuration object to pass into an instance of the `SdlManager` class and then start it. This configuration object requires a `LifecycleConfig` object which contains the majority of the settings you need to set in order for the app to function. You will want to use the following methods to get started: `setAppId`, `setAppName`, `setLanguageDesired`, `setAppTypes`, and `setTransportConfig`. These configuration objects support method chaining, which allow you to create the following example configuration object:

```
const lifecycleConfig = new SDL.manager.LifecycleConfig()
    .setAppId('hello-js')
    .setAppName('Hello JS')
    .setLanguageDesired(SDL.rpc.enums.Language.EN_US)
    .setAppTypes([
        SDL.rpc.enums.AppHMIType.DEFAULT,
    ]);
```

> **NOTE**
>
> For WebEngine apps, most of this configuration will happen in the `manifest.js` file and does not need to be duplicated here.

Configure Module Support

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

BLOCKING BY VERSION

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

```
lifecycleConfig.setMinimumProtocolVersion(new SDL.util.Version(3, 0, 0));
lifecycleConfig.setMinimumRpcVersion(new SDL.util.Version(4, 0, 0));
```

BLOCKING BY VEHICLE TYPE

If you are blocking by vehicle type and you are connected over RPC v7.1+, your app icon will never appear on the head unit's screen. If you are connected over RPC v7.0 or below, it will appear and then quickly disappear. To implement this type of blocking, you need to set up the SDLManager. You will then implement the optional `onSystemInfoReceived` method and return `true` if you want to continue the connection and `false` if you wish to disconnect.

Transport Configuration

The transport configuration will depend on the environment you're using. For example, you can make a TCP connection with Node.js, but not with vanilla JS. See below for example transport configurations.

NODE.JS TCP TRANSPORT

```
lifecycleConfig.setTransportConfig(new SDL.transport.TcpClientConfig(HOST, PORT));
```

For a WebSocket server connection, the developer is expected to set up the server component, and pass in incoming client connections to the `WebSocketServerConfig`. The `ws` node module provides the necessary object to the config.

NODE.JS WEBSOCKET SERVER

```
const WS = require('ws');
const PORT = 3000;

// create a WebSocket Server
const appWebSocketServer = new WS.Server({
    port: PORT,
});
console.log(`WebSocket Server listening on port ${PORT}`);

appWebSocketServer.on('connection', (connection) => {
    // app setup goes here
    lifecycleConfig.setTransportConfig(
        new SDL.transport.WebSocketServerConfig(
            connection,
            CONNECTION_LOST_TIMEOUT // connection timeout in milliseconds (default
is 60 seconds)
        )
    );
});
```

VANILLA JS WEBSOCKET CLIENT

```
lifecycleConfig.setTransportConfig(new SDL.transport.WebSocketClientConfig(HOST,
PORT));
```

Additional Configuration Options

There are several additional basic configuration options to set up your app, like the app name and icon.

App Icon

An app icon can be set in the `LifecycleConfig` to automatically upload and set the icon image. Note that although the implementation of retrieving files are different between the JS browser and Node.js environments, the developer can use the same API in both cases,

and the SDL library will cover the implementation details for the developer depending on which build they are using.

```
const filePath = './app_icon.png';
const file = new SDL.manager.file.filetypes.SdlFile()
    .setName('AppIcon')
    .setFilePath(filePath)
    .setType(SDL.rpc.enums.FileType.GRAPHIC_PNG)
    .setPersistent(true);

lifecycleConfig.setAppIcon(file);
```

In this case, the code snippet expects there to be an `app_icon.png` file present in the same directory for the app icon.

Listening for RPC notifications and events

You can listen for specific events using the `LifecycleConfig`'s `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 `OnHMIStatus` in the following example.

```
lifecycleConfig.setRpcNotificationListeners({
    [SDL.rpc.enums.FunctionID.OnHMIStatus]: (onHmiStatus) => {
        // HMI Level updates
        const hmiLevel = onHmiStatus.getHmiLevel();
        console.log("Current HMI Level: ", hmiLevel);
    }
});
```

It is recommended to use this method over the `SdlManager.addRpcListener` method for the `OnHMIStatus` RPC, or any RPC Notifications that your app cannot afford to miss during the initial connection.

Setting Up the SDL Manager

After creating the `LifecycleConfig`, it can be set into the `AppConfig` and then passed into the `SdlManager`. The following snippet will set up the `SdlManager` and start it up. A listener is attached to the manager listener to let you know when there is a connection and the managers are ready.

```
const appConfig = new SDL.manager.AppConfig()
    .setLifecycleConfig(lifecycleConfig);

const managerListener = new SDL.manager.SdlManagerListener()
    .setOnStart((sdlManager) => {
        // managers are ready
    })
    .setOnError((sdlManager, info) => {
        console.error('Error from SdlManagerListener: ', info);
    })
    .setOnSystemInfoReceived((systemInfo) => {
        console.log(`Connected to system ${systemInfo}`);
        return true;
    })
    .setManagerShouldUpdateLifecycleToLanguage((language, hmiLanguage) => {
        return new SDL.manager.lifecycle.LifecycleConfigurationUpdate();
    });

const sdlManager = new SDL.manager.SdlManager(appConfig, managerListener)
    .start();
```

Configuring the WebEngine App HTML File

For WebEngine apps, there are slight modifications for integrating the library, such as importing the manifest file and passing it into the `LifecycleConfig.loadManifest` method. Additionally, since the data for the connection information is part of the URL query, the `WebSocketClientConfig` class requires no arguments, and the library will read the URL query values instead. The resulting `index.html` may look something like this as a result:

```
<html>
  <head>
    <script src='./SDL.min.js'></script>
  </head>
  <body>
    <script type='module'>
      import sdl_manifest from './manifest.js';

      const lifecycleConfig = new SDL.manager.LifecycleConfig()
        .loadManifest(sdl_manifest)
        .setLanguageDesired(SDL.rpc.enums.Language.EN_US);

      lifecycleConfig.setTransportConfig(new
SDL.transport.WebSocketClientConfig());

      lifecycleConfig.setRpcNotificationListeners({
        [SDL.rpc.enums.FunctionID.OnHMIStatus]: (onHmiStatus) => {
          // HMI Level updates
          const hmiLevel = onHmiStatus.getHmiLevel();
          console.log("Current HMI Level: ", hmiLevel);
        }
      });

      const appConfig = new SDL.manager.AppConfig()
        .setLifecycleConfig(lifecycleConfig);

      const managerListener = new SDL.manager.SdlManagerListener()
        .setOnStart((sdlManager) => {
          // managers are ready
        })
        .setOnError((sdlManager, info) => {
          console.error('Error from SdlManagerListener: ', info);
        })
        .setOnSystemInfoReceived((systemInfo) => {
          console.log(`Connected to system ${systemInfo}`);
          return true;
        })
        .setManagerShouldUpdateLifecycleToLanguage((language, hmiLanguage)
=> {
          return new SDL.manager.lifecycle.LifecycleConfigurationUpdate();
        });

      const sdlManager = new SDL.manager.SdlManager(appConfig,
managerListener)
        .start();
    </script>
  </body>
</html>
```

Where to Go From Here

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

Connecting to an Infotainment System

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

Connecting to an Emulator

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

Getting the IP Address and Port

GENERIC SDL CORE

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

MANTICORE

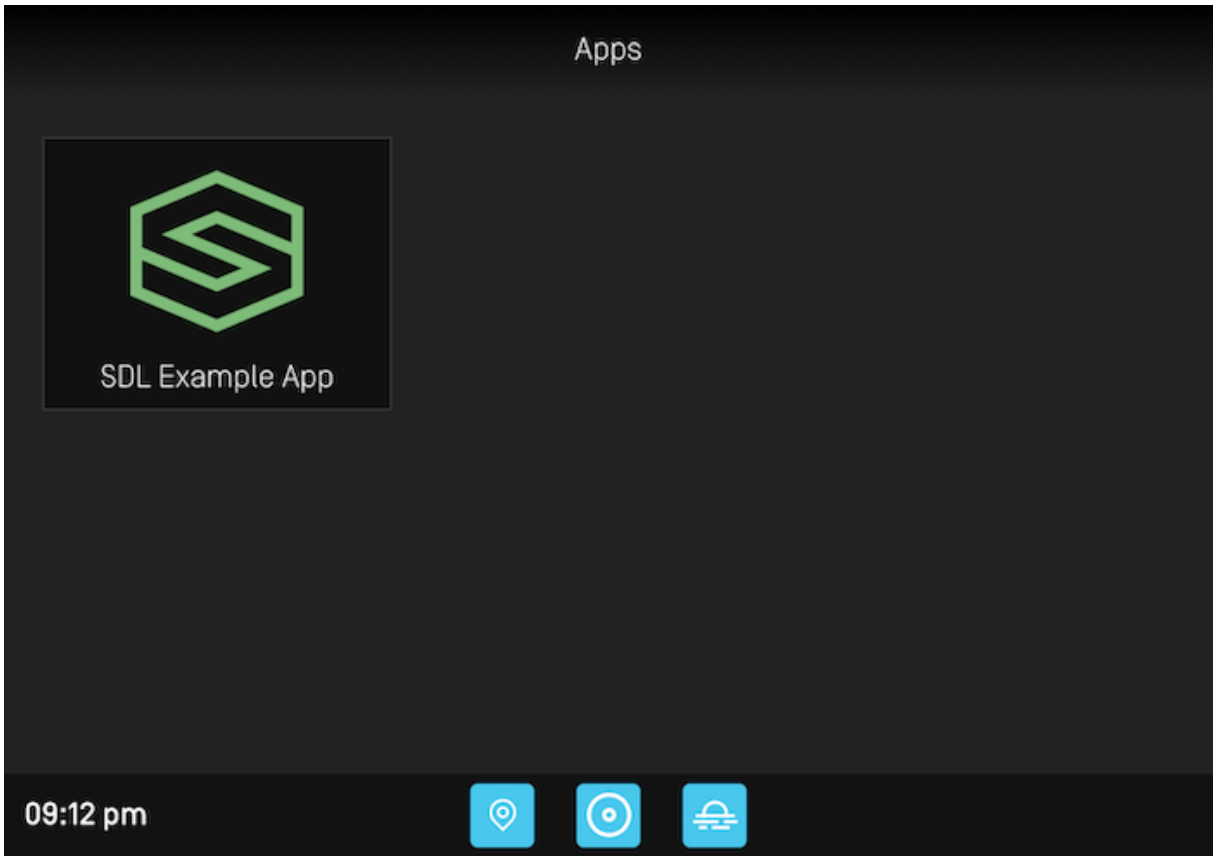
Once you launch an instance of Manticore, you will be given an IP address and port number that you can use to configure your TCP connection.

Setting the IP Address and Port

```
const lifecycleConfiguration = new
SDL.manager.LifecycleConfig().setTransportConfig(new
SDL.transport.TcpClientConfig(<IP Address>, <PORT>));
```

Running the SDL App

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



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

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

Troubleshooting

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

Your SDL cloud/webengine 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>"]
}
```

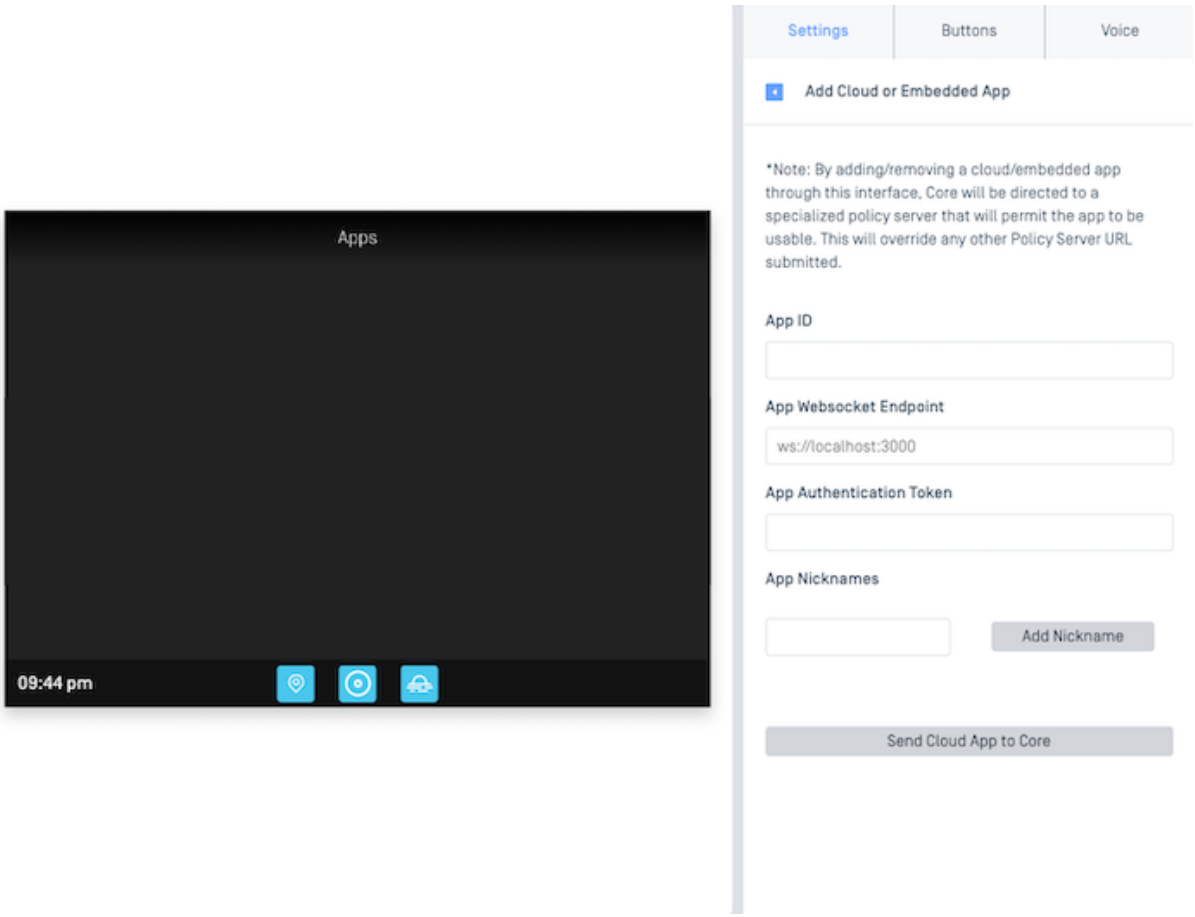
NOTE

The `<app name>` value in `"nicknames"` must match the app name value used in Integration Basics when implementing the SDL manager.

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

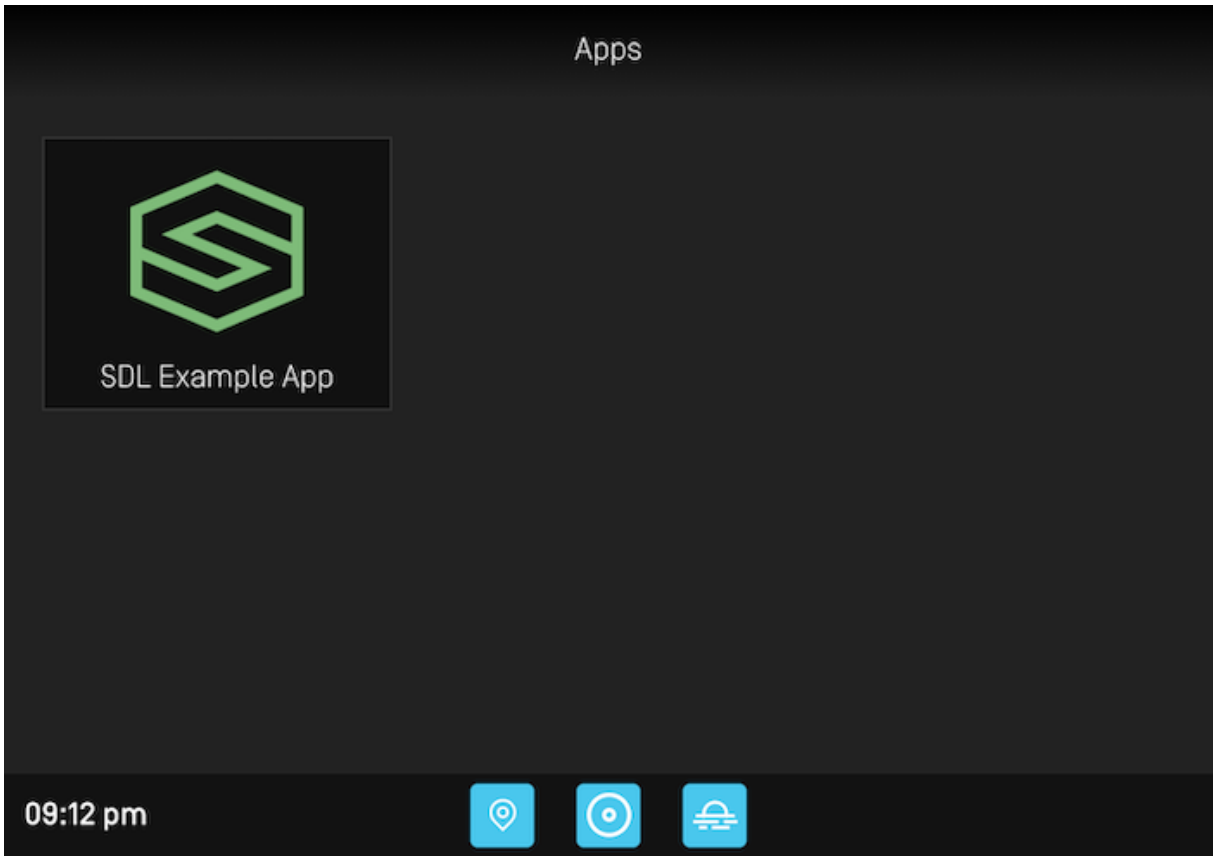
Manticore

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.



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 `SDL.manager.LifecycleConfig`. 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 `SDL.manager.LifecycleConfig`. 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.

```
const headUnitLanguage =
sdlManager.getRegisterAppInterfaceResponse().getLanguage();
const 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 LifecyleConfig.
2. Implement the `sdlManagerListener`'s `managerShouldUpdateLifecycle(language, hmiLanguage)` method. If the module's current HMI language or voice recognition (VR) language is different from the app's default language, the listener will be called with the module's current HMI and/or VR language. Return a `LifecycleConfigurationUpdate` with the new `appName` and/or `ttsName`.

```
managerShouldUpdateLifecycle(language, hmiLanguage) {
    let isUpdateNeeded = false;
    let appName = APP_NAME;
    let ttsName = APP_NAME;
    switch (language) {
        case SDL.rpc.enums.Language.ES_MX:
            isUpdateNeeded = true;
            ttsName = APP_NAME_ES;
            break;
        case SDL.rpc.enums.Language.FR_CA:
            isUpdateNeeded = true;
            ttsName = APP_NAME_FR;
            break;
        default:
            break;
    }
    switch (hmiLanguage) {
        case SDL.rpc.enums.Language.ES_MX:
            isUpdateNeeded = true;
            appName = APP_NAME_ES;
            break;
        case SDL.rpc.enums.Language.FR_CA:
            isUpdateNeeded = true;
            appName = APP_NAME_FR;
            break;
        default:
            break;
    }
    if (isUpdateNeeded) {
        const chunks = [new
SDL.rpc.structs.TTSChunk().setText(ttsName).setType(SDL.rpc.enums.SpeechCapabi

        return new SDL.manager.lifecycle.LifecycleConfigurationUpdate(appName, null,
chunks, null);
    } else {
        return 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 `LifecycleConfig`'s `setRpcNotificationListeners`.

```
function onHmiStatusListener (onHmiStatus) {
    const hmiLevel = onHmiStatus.getHmiLevel();
    if (hmiLevel === SDL.rpc.enums.HMILevel.HMI_FULL) {
        // now in HMI FULL
    }
}

lifecycleConfig.setRpcNotificationListeners({
    [SDL.rpc.enums.FunctionID.OnHMIStatus]: onHmiStatusListener
});
```

Permission Manager

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

Checking Current Permissions of a Single RPC

```
const isAllowed =
sdlManager.getPermissionManager().isRpcAllowed(SDL.rpc.enums.FunctionID.Show)


const isParameterAllowed =
sdlManager.getPermissionManager().isPermissionParameterAllowed(SDL.rpc.enums
 SDL.rpc.messages.GetVehicleData.KEY_RPM);
```

Checking Current Permissions of a Group of RPCs

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

```
const permissionElements = [];
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.Show,
null));
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.GetVehicleD
 [SDL.rpc.messages.GetVehicleData.KEY_RPM,
SDL.rpc.messages.GetVehicleData.KEY_SPEED]));

const groupStatus =
sdlManager.getPermissionManager().getGroupStatusOfPermissions(permissionElem


switch (groupStatus) {
    case SDL.manager.permission.enums.PermissionGroupStatus.ALLOWED:
        // Every permission in the group is currently allowed
        break;
    case SDL.manager.permission.enums.PermissionGroupStatus.DISALLOWED:
        // Every permission in the group is currently disallowed
        break;
    case SDL.manager.permission.enums.PermissionGroupStatus.MIXED:
        // Some permissions in the group are allowed and some disallowed
        break;
    case SDL.manager.permission.enums.PermissionGroupStatus.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 you want to get a more detailed result about the status of every permission or parameter in the group, you can use the `getStatusOfPermissions` method.

```
const permissionElements = [];
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.Show,
null));
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.GetVehicleD
 [SDL.rpc.messages.GetVehicleData.KEY_RPM,
SDL.rpc.messages.GetVehicleData.KEY_AIRBAG_STATUS]));

const status =
sdlManager.getPermissionManager().getStatusOfPermissions(permissionElements);

if (status[SDL.rpc.enums.FunctionID.GetVehicleData].getIsRpcAllowed()){
    // GetVehicleData RPC is allowed
}

if (status[SDL.rpc.enums.FunctionID.GetVehicleData].getAllowedParameters()
[SDL.rpc.messages.GetVehicleData.KEY_RPM]){
    // 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 `SDL.manager.permission.enums.PermissionGroupType.ANY`. If you only want to be notified when all of the RPCs in the group are allowed, or go from allowed to some/all not allowed, set the `groupType` to `SDL.manager.permission.enums.PermissionGroupType.ALL_ALLOWED`.

```
const permissionElements = [];
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.Show,
null));
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.GetVehicleD
 [SDL.rpc.messages.GetVehicleData.KEY_RPM,
SDL.rpc.messages.GetVehicleData.KEY_AIRBAG_STATUS]));

const listenerId =
sdlManager.getPermissionManager().addListener(permissionElements,
SDL.manager.permission.enums.PermissionGroupType.ANY, function
(allowedPermissions, permissionGroupStatus) {
    if
(allowedPermissions[SDL.rpc.enums.FunctionID.GetVehicleData].getIsRpcAllowed())
{
        // GetVehicleData RPC is allowed
    }

    if
(allowedPermissions[SDL.rpc.enums.FunctionID.GetVehicleData].getAllowedParamet
[SDL.rpc.messages.GetVehicleData.KEY_RPM]){
        // 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(listenerUuid);
```

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.

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnHMIStatus, onHmiStatus
=> {
    const streamingState = onHmiStatus.getAudioStreamingState();
});
```

The code snippet above will get the AudioStreamingState which reflects the HMI's ability to stream audio. However, the JavaScript Suite does not yet support audio and video streaming. This will be addressed in a future version.

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.

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnHMIStatus, onHmiStatus
=> {
    const systemContext = onHmiStatus.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.

```
// sdl_javascript_suite v1.1+
async function send () {
    const response = await sdlManager.sendRpcResolve(alert);
    if (!response.getSuccess()) {
        // The request was not successful. Check the response's result code for more
information
        return;
    }
    // The request was successful
}
send().catch(err => {
    // thrown exceptions will be caught here
    // catch exceptional behavior in a parent function instead of at the RPC sending
level
});

// Pre sdl_javascript_suite v1.1
(async function () {
    const response = await sdlManager.sendRpc(request);
    if (!response.getSuccess()) {
        // The request was not successful. Check the response's result code or catch
and log the Promise error for more information
        return;
    }
    // The request was successful
})();
```

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.

```
const rpcSpecVersion =
sdlManager.getRegisterAppInterfaceResponse().getSdlMsgVersion();
```

Example Apps

The JavaScript Suite repository on GitHub provides example apps for both the browser and for NodeJS. This includes a WebEngine app, a WebSocket client app, a WebSocket server app, and a TCP client app. The examples in the folders already come with their own SDL library build files. Check each example app's `readme.md` file for more information on how to run the respective app.

Troubleshooting

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

TCP Debug Transport

1. Make sure that your `HOST` and `PORT` environment variables are set to match the machine running SDL Core.
2. Make sure there is no firewall blocking the incoming port `12345` on the machine or VM running SDL Core. Also, make sure your firewall allows that outgoing port.
3. There are different network configurations needed for different virtualization software (VirtualBox, VMware, etc). Make sure yours is set up correctly. Or use Manticore.

Websocket Transport

1. Make sure that the policy table of SDL Core has the correct app IDs and nicknames as well as `enabled=true`.

2. Make sure that the cloud endpoint and cloud transport type provided to SDL Core are correct and reachable by SDL core. There are different network configurations needed for different virtualization software (VirtualBox, VMware, etc). Make sure yours is set up correctly.

WebEngine Transport

1. Make sure that the `manifest.js` has provided all necessary fields and that the information is correct.
2. If you're unable to install your app from a cloud app store, make sure that the app has been compressed to a file archive that can be retrieved via the download URL you provided.

Connecting to an Infotainment System

Connecting as a WebSocket Client

For vanilla JavaScript SDL apps, connecting as a WebSocket client requires a version of SDL Core that can accept incoming WebSocket connections (at least v6.1.0). If you are using Manticore to test your app, note that it currently does not support WebSocket connections.

A workaround to this limitation is to use a proxy program for your app to connect with which modifies the incoming WebSocket connection into a TCP connection. The proxy program then connects to SDL Core on the app's behalf and passes through your transport data. A Java program that does exactly this is available in the repository's JavaScript example folder called `proxy.jar`. Check the example app's `readme.md` for how to run it. Please check the Connecting to an Infotainment System guide for more detailed instructions on how to get the emulator's IP address and port number.

This workaround for older versions of Core is also necessary for WebEngine apps.

Connecting as a WebSocket Server

SDL Core acts as the WebSocket client in this case. The information about your app and how Core should connect to it must go into the policy table. Check the Connecting to an Infotainment System guide's **Configuring the Connection** section for how to set up your policy table to point to your app

The following snippet is a truncated version of what is needed to set up the WebSocket server to accept and pass connections to the SDL library. This example uses the `ws` npm module for WebSocket connections. Refer to the integration basics guide for the full integration setup.

```
const SDL = require('./SDL.min.js');
const WS = require('ws');
const PORT = 3000;

// create a WebSocket Server
const appWebSocketServer = new WS.Server({
    port: PORT,
});
console.log(`WebSocket Server listening on port ${PORT}`);

// Event listener for incoming WebSocket connections
appWebSocketServer.on('connection', (connection) => {
    ...
    /* truncated snippet to show only the transport configuration setup */
    /* each new connection corresponds to a new instance of your app */
    lifecycleConfig.setTransportConfig(
        new SDL.transport.WebSocketServerConfig(
            connection
        )
    );
    ...
});
```

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/Methods

PARAMETERS/METHODS	DESCRIPTION	RPC VERSION
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).	RPC v6.0+
getHmiZoneCapabilities()	Specifies HMI Zones in the vehicle. There may be a HMI available for back seat passengers as well as front seat passengers.	RPC v1.0+
getSpeechCapabilities()	Contains information about TTS capabilities on the SDL platform. Platforms may support text, SAPI phonemes, LH PLUS phonemes, pre-recorded speech, and silence.	RPC v1.0+
getPrerecordedSpeechCapabilities()	A list of pre-recorded sounds you can use in your app. Sounds may include a help, initial, listen, positive, or a negative jingle.Currently only available in the SDL_iOS and SDL JavaScript libraries	RPC v3.0+
getVrCapabilities()	The voice-recognition capabilities of the connected SDL platform. The platform may be able to recognize spoken text in the current language.	RPC v1.0+

PARAMETERS/METHODS	DESCRIPTION	RPC VERSION
getAudioPassThruCapabilities()	Describes the sampling rate, bits per sample, and audio types available.	RPC v2.0+
getPcmStreamCapabilities()	Describes different audio type configurations for the audio PCM stream service, e.g. {8kHz,8-bit,PCM}.	RPC v4.1+
getHmiCapabilities()	Returns whether or not the app can support built-in navigation and phone calls.	RPC v3.0+
SystemCapabilityType.APP_SERVICES	Describes the capabilities of app services including what service types are supported and the current state of services.	RPC v5.1+
SystemCapabilityType.NAVIGATION	Describes the built-in vehicle navigation system's APIs.	RPC v4.5+
SystemCapabilityType.PHONE_CALL	Describes the built-in phone calling capabilities of the IVI system.	RPC v4.5+
SystemCapabilityType.VIDEO_STREAMING	Describes the abilities of the head unit to video stream projection applications.	RPC v4.5+
SystemCapabilityType.REMOTE_CONTROL	Describes the abilities of an app to control built-in aspects of the IVI system.	RPC v4.5+
SystemCapabilityType.SEAT_LOCATION	Describes the positioning of each seat in a vehicle	RPC v6.0+

Deprecated Properties

The following properties are deprecated on SDL JavaScript 1.0 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
getDisplayCapabilities()	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.
getDefaultMainWindowCapability().getButtonCapabilities()	A list of available buttons and whether the buttons support long, short and up-down presses.
getDefaultMainWindowCapability().getSoftButtonCapabilities()	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.
getPresetBankCapabilities()	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.

```
const field =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().getIma
[index];
const 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 and Subscribing System Capabilities

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

Determining Support for System Capabilities

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

```
const navigationSupported = sdlManager.getSystemCapabilityManager().isCapabilitySupported(SDL.rpc.enums.Sys
```

Manual Querying for System Capabilities

Most head units provide features that your app can use: making and receiving phone calls, an embedded navigation system, video and audio streaming, as well as supporting app services. To pull information about this capability, use the `SystemCapabilityManager` to

query the head unit for the desired capability. If a capability is unavailable, the query will return `null`.

```
const appServicesCapabilities = await
sdlManager.getSystemCapabilityManager().updateCapability(SDL.rpc.enums.SystemC

if (appServicesCapabilities !== null) {
    // Capability retrieved
} else {
    // Handle Error
}
```

Subscribing to System Capabilities (RPC v5.1+)

In addition to getting the current system capabilities, it is also possible to subscribe for updates when the head unit capabilities change.

> **NOTE**
>
> If `supportsSubscriptions === false`, you can still subscribe to capabilities, however, you must manually poll for new capability updates using `getCapability(type)`. All subscriptions will be automatically updated when that method returns a new value.
>
> The `DISPLAYS` type can be subscribed on all SDL versions.

CHECKING IF THE HEAD UNIT SUPPORTS SUBSCRIPTIONS

The `supportsSubscriptions` method currently is not supported by the JavaScript Suite. This will be addressed in a future release.

SUBSCRIBE TO A CAPABILITY

```
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(SDL.rpc.e
 function (capability) {
    // This listener is now subscribed to AppServicesCapabilities
    if (capability instanceof SDL.rpc.structs.AppServicesCapabilities) {
        // Got an AppServicesCapabilities struct
    }
})
```

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, use `ScreenManager.changeLayout()`. This guide requires SDL JavaScript Suite version 1.2. If using an older version, use the `SetDisplayLayout` RPC.

NOTE

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

```
const templateConfiguration = new SDL.rpc.structs.TemplateConfiguration()
    .setTemplate(SDL.rpc.enums.PredefinedLayout.GRAPHIC_WITH_TEXT);

const success = await
sdlManager.getScreenManager().changeLayout(templateConfiguration);
if (success) {
    console.log('Layout set successfully');
} else {
    console.log('Layout not set successfully');
}
```

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

```
sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setTextField1('Line of Text');
// The promise returned by changeLayout will not resolve because it is part of a
batch update, and the await operator should be avoided as a result
sdlManager.getScreenManager().changeLayout(templateConfiguration);
sdlManager.getScreenManager().setPrimaryGraphic(artwork);
const success = await sdlManager.getScreenManager().commit();
if (success) {
    console.log('Text, Graphic, and Template changed successful');
}
```

When changing screen layouts and template data (for example, to show a weather hourly data screen vs. a daily weather screen), it is recommended to encapsulate these updates

into a class or method. Doing so is a good way to keep SDL UI changes organized. Below is a generic example.

Screen Change Example Code

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

Screen Change Example Interface

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

```
class CustomSdlScreen {
    constructor (sdlManager) {
        this.sdlManager = sdlManager;
    }
    showScreen () {
        // stub
    }
}
```

Screen Change Example Implementations

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

For the example below, the `HomeScreen` class will inherit the `CustomSDLScreen` interface and will have a property of type `HomeDataViewModel`. The screen manager will

change its text fields based on the view model's data. In addition, the home screen will also create a navigation button to open the `ButtonSDLScreen` when pressed.

```
class HomeSdlScreen extends CustomSdlScreen {
    constructor (sdlManager) {
        super(sdlManager);
        this.buttonScreen = new ButtonSdlScreen(sdlManager);
        this.homeDataViewModel = new HomeDataViewModel(); // holds plain object
data
    }

    showScreen () {
        const screenManager = this.sdlManager.getScreenManager();
        // Batch Updates
        screenManager.beginTransaction();
        // Change template to Graphics With Text and Soft Buttons
        screenManager.changeLayout(new SDL.rpc.structs.TemplateConfiguration()

.setTemplate(SDL.rpc.enums.PredefinedLayout.GRAPHIC_WITH_TEXT_AND_SOFTBU

        // Assign text fields to view model data
        screenManager.setTextField1(this.homeDataViewModel.text1);
        screenManager.setTextField2(this.homeDataViewModel.text2);
        screenManager.setTextField3(this.homeDataViewModel.text3);
        screenManager.setTextField4(this.homeDataViewModel.text4);
        // Create and assign a button to navigate to the ButtonSdlScreen
        const navigationButton = new
SDL.manager.screen.utils.SoftButtonObject('ButtonSdlScreen', [new
SDL.manager.screen.utils.SoftButtonState('ButtonSdlScreen', 'Button Screen')],
'ButtonSdlScreen', (id, rpc) => {
            if (rpc instanceof SDL.rpc.messages.OnButtonPress) {
                this.buttonScreen.showScreen();
            }
        });
        screenManager.setSoftButtonObjects([navigationButton]);
        screenManager.commit();
    }
}
```

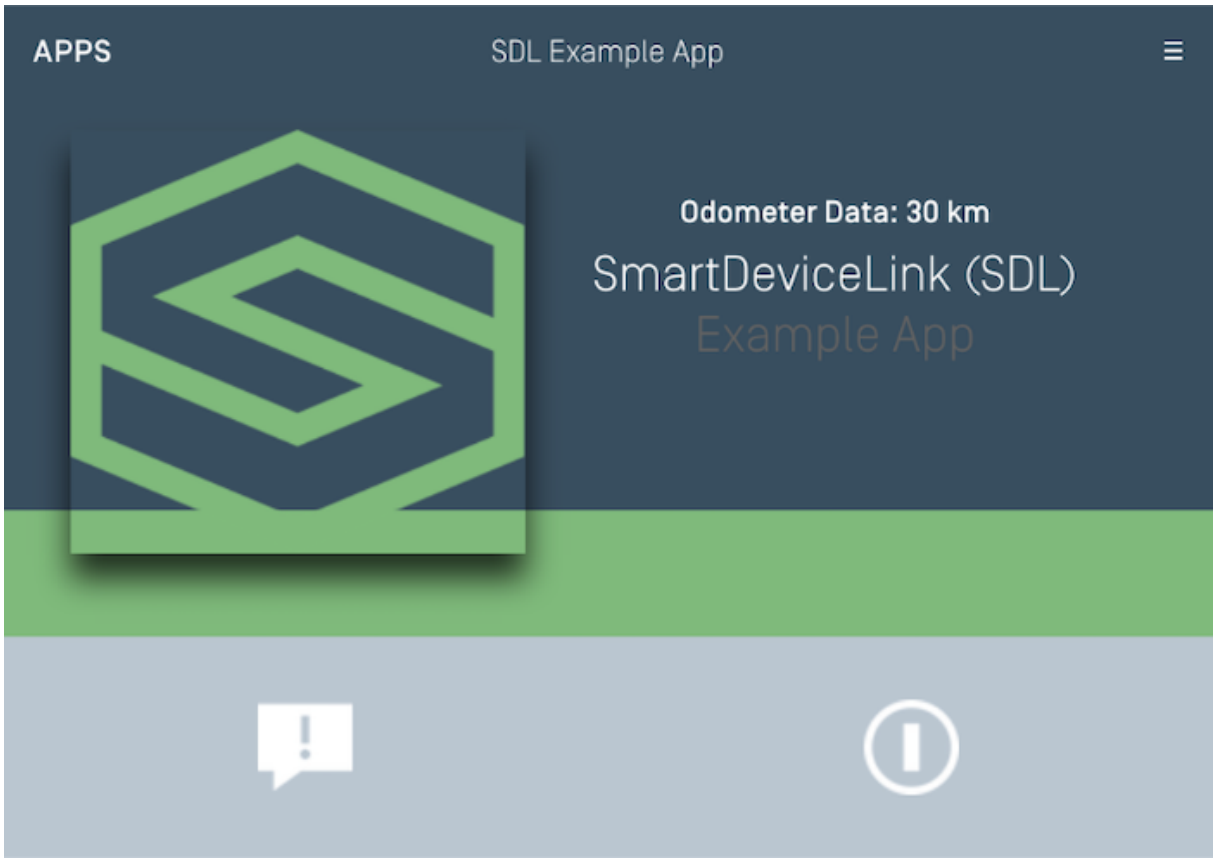
The `ButtonSDLScreen` follows the same patterns as the `HomeSDLScreen` but has minor implementation differences. The screen's view model `ButtonDataViewModel` contains properties unique to the `ButtonSDLScreen` such as text fields and an array of soft button objects. It also changes the template configuration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

```
class ButtonSdlScreen extends CustomSdlScreen {
    constructor (sdlManager) {
        super(sdlManager);
        this.buttonDataViewModel = new ButtonDataViewModel(); // holds plain object
data
    }
    showScreen () {
        const screenManager = this.sdlManager.getScreenManager();
        // Batch Updates
        screenManager.beginTransaction();
        // Change template to Tiles Only
        screenManager.changeLayout(new SDL.rpc.structs.TemplateConfiguration()
            .setTemplate(SDL.rpc.enums.PredefinedLayout.TILES_ONLY));
        // Assign soft button objects to view model buttons array
        screenManager.setSoftButtonObjects(this.buttonDataViewModel.buttons);
        screenManager.commit();
    }
}
```

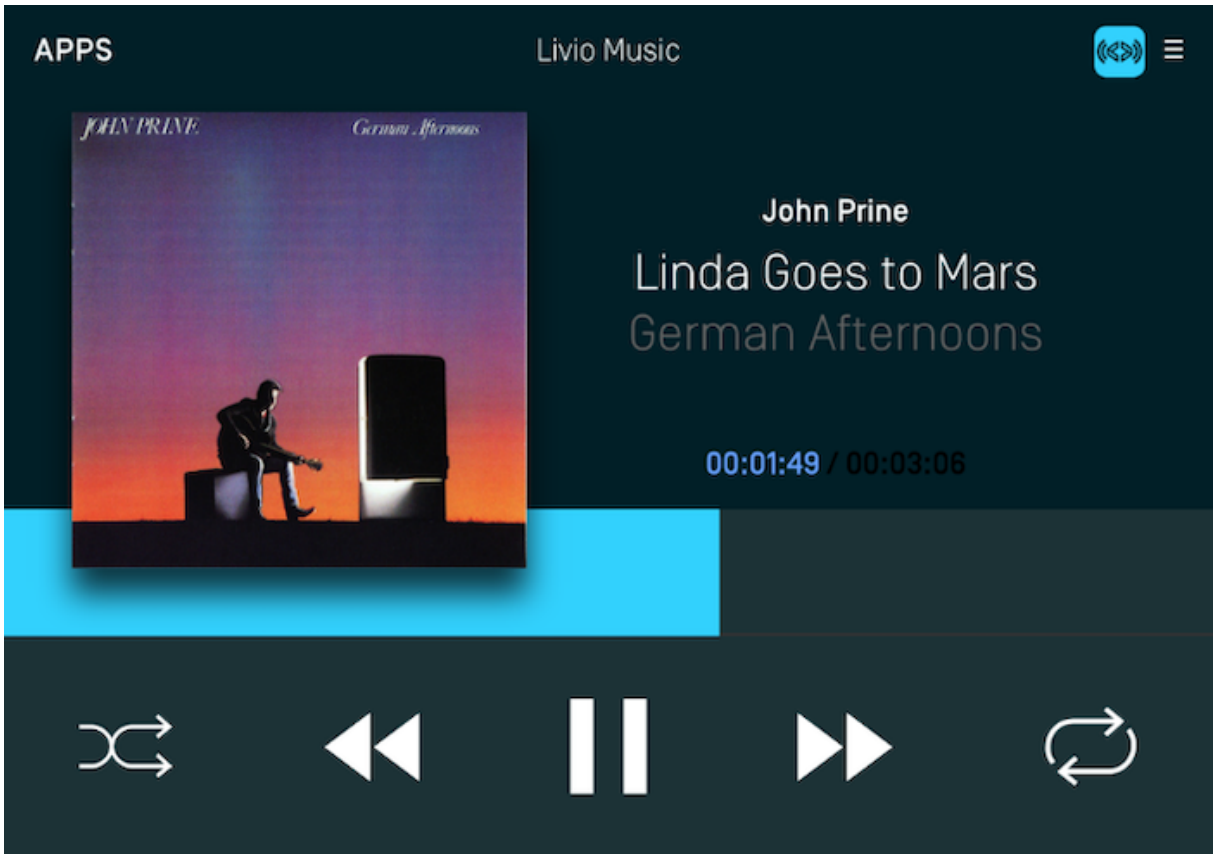
Available Templates

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

MEDIA



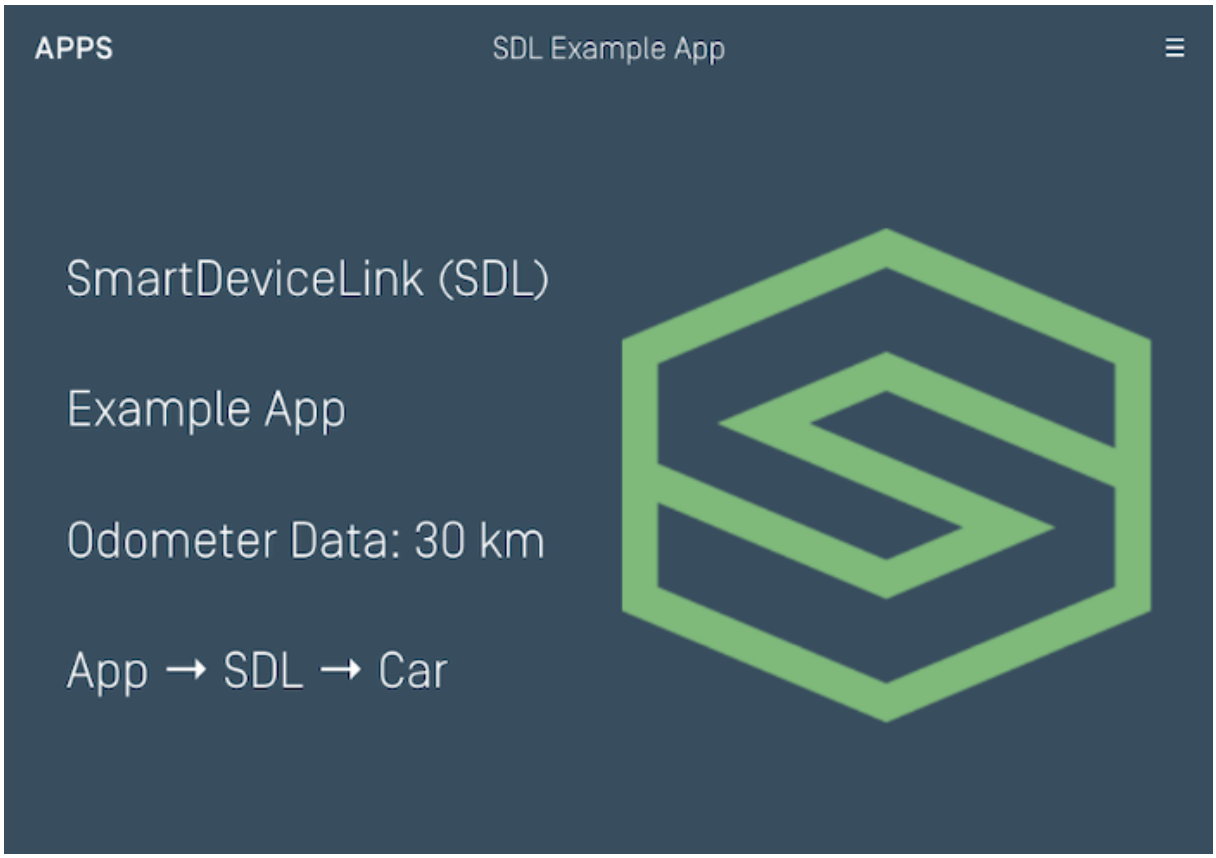
MEDIA (WITH A PROGRESS BAR)


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

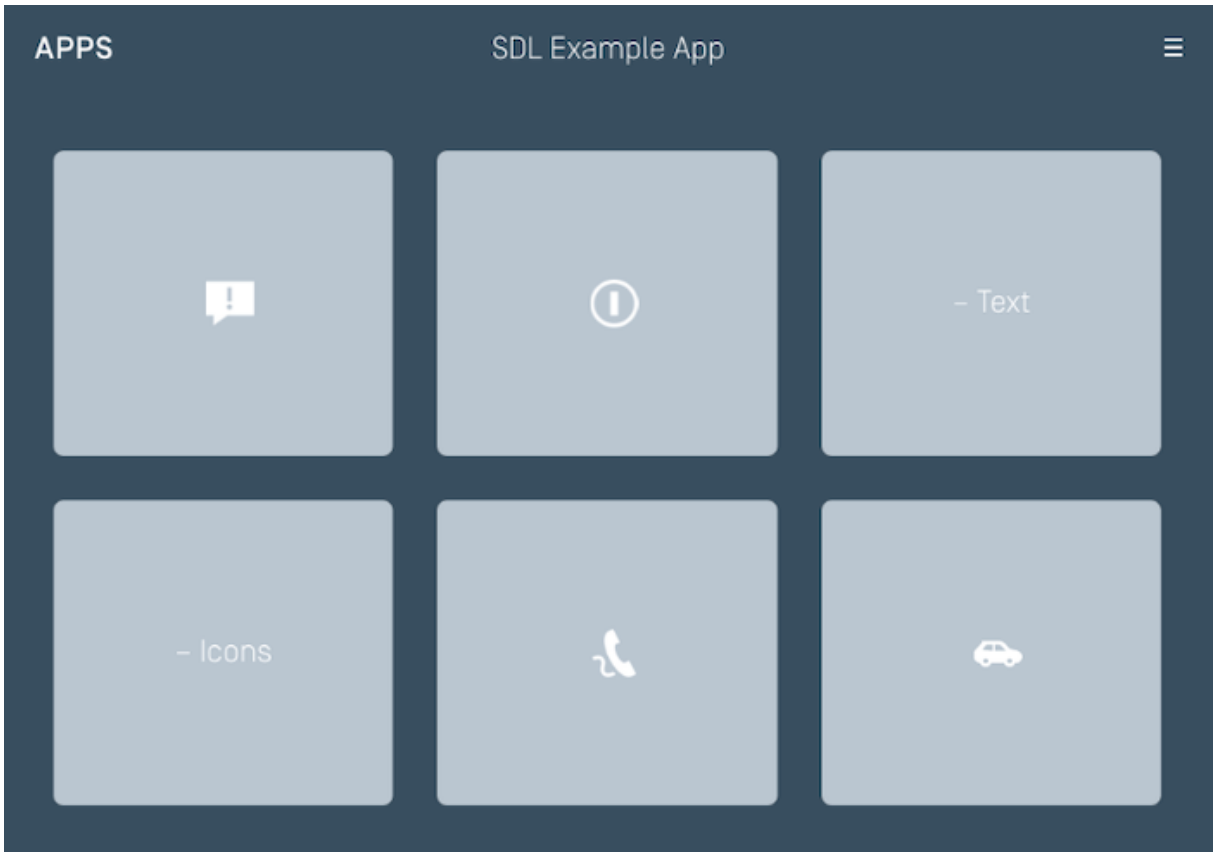
NON-MEDIA

GRAPHIC WITH TEXT

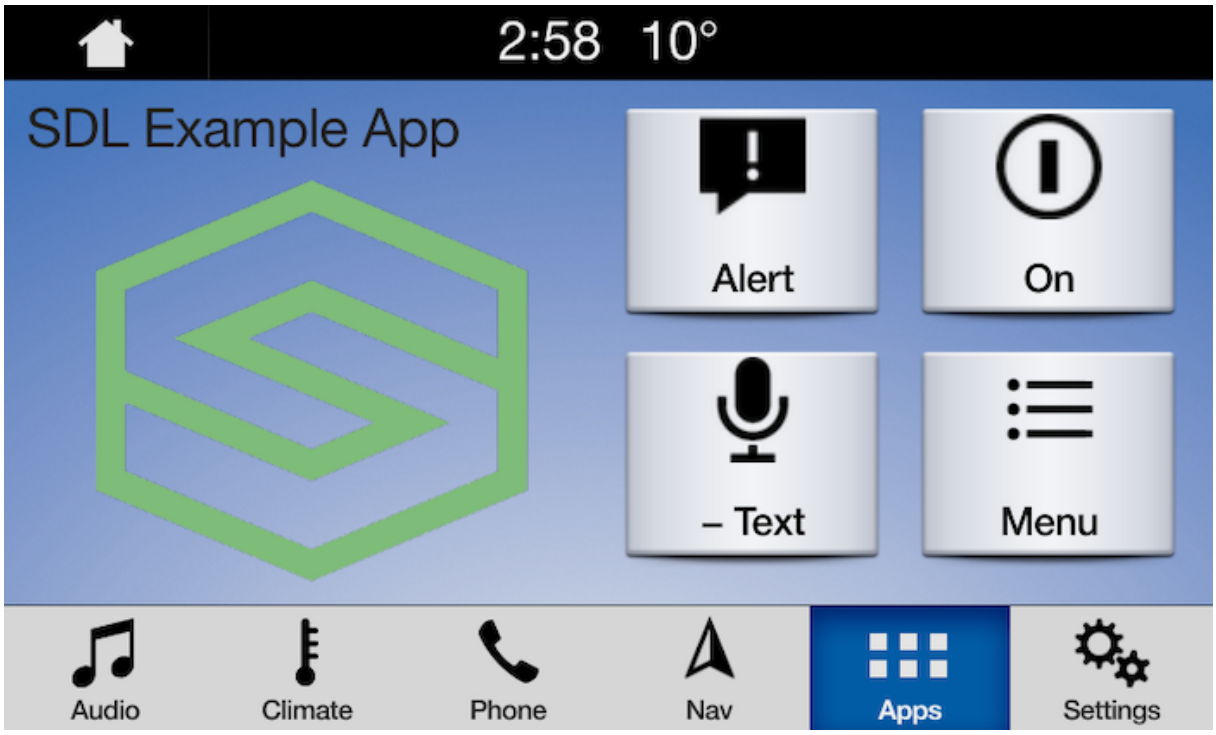
TEXT WITH GRAPHIC


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

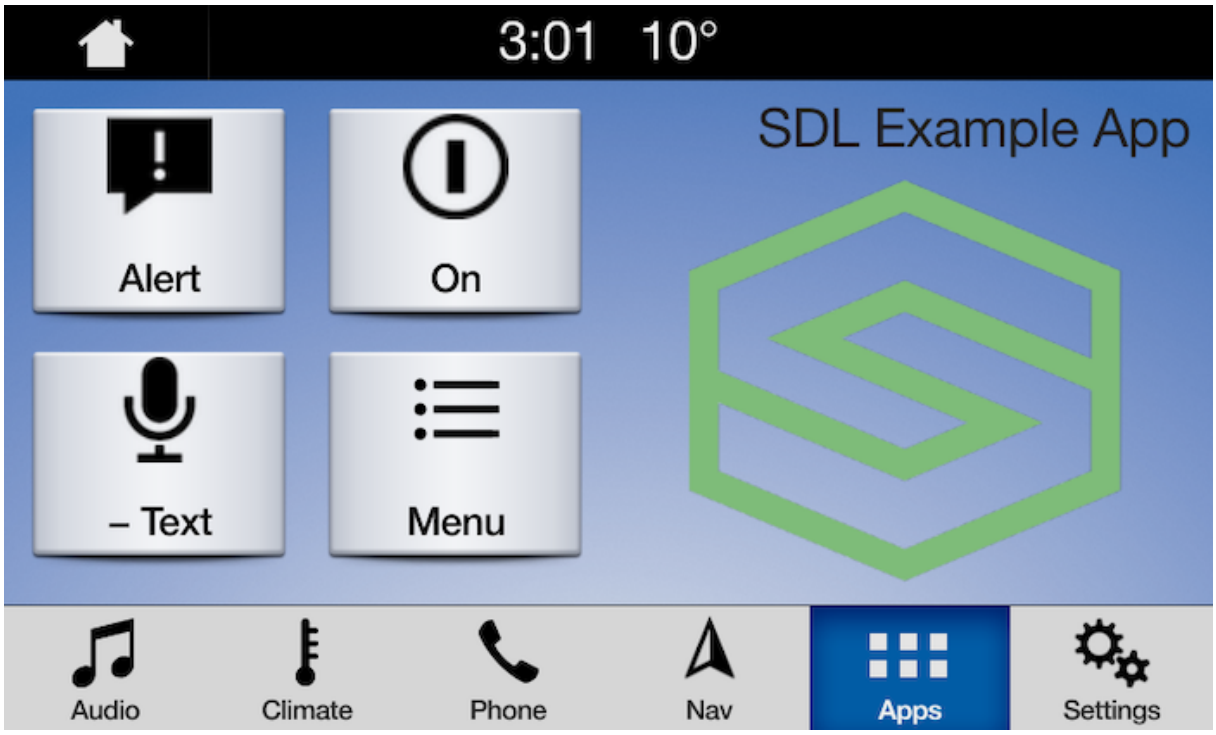
TILES ONLY



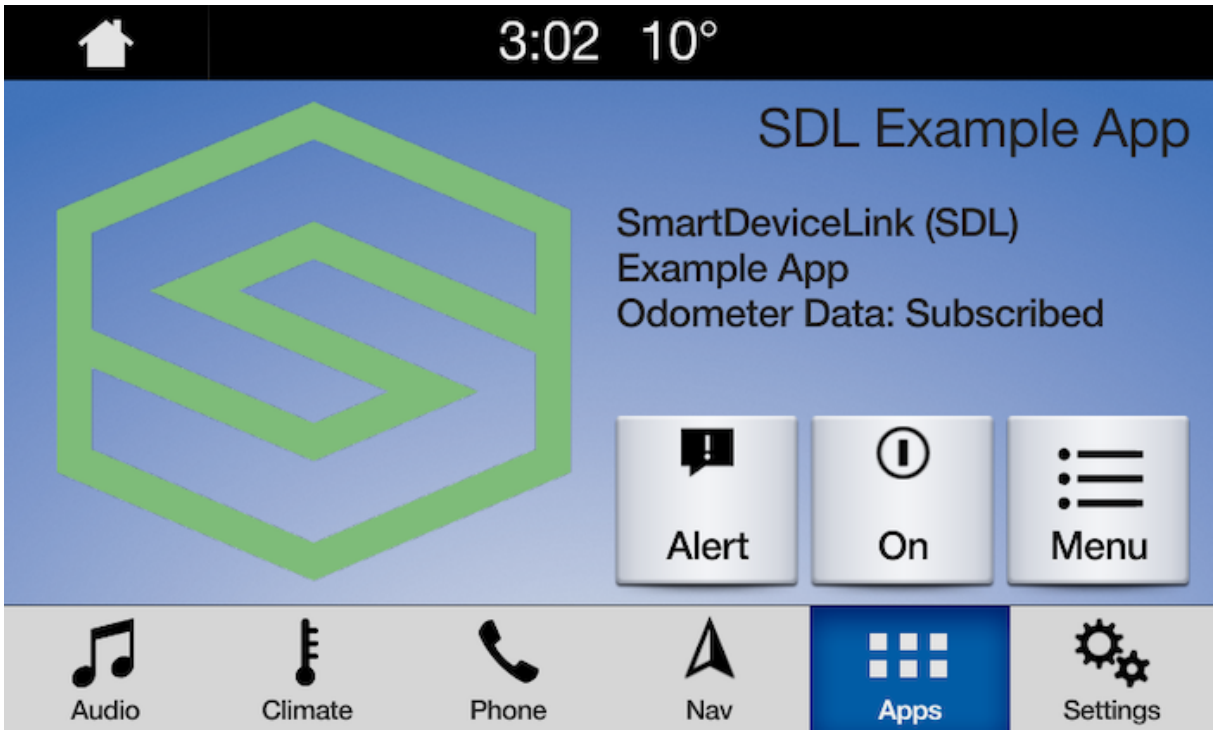
GRAPHIC WITH TILES



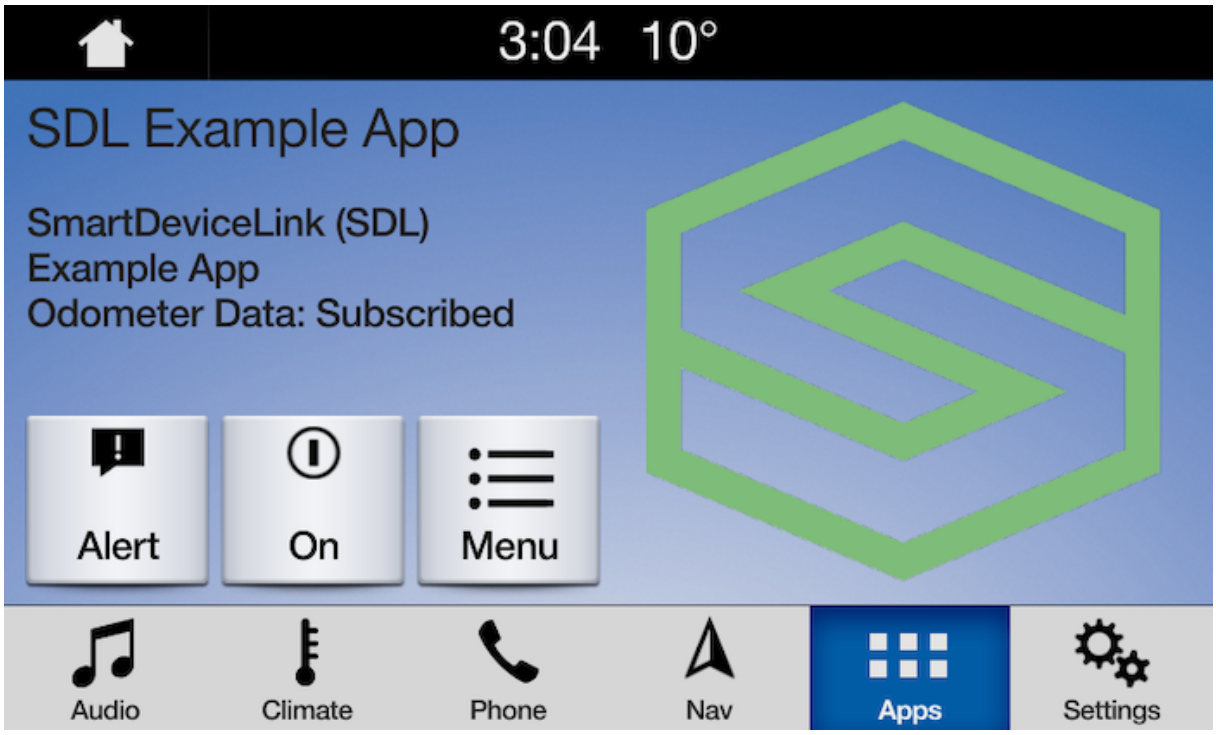
TILES WITH GRAPHIC



GRAPHIC WITH TEXT AND SOFT BUTTONS

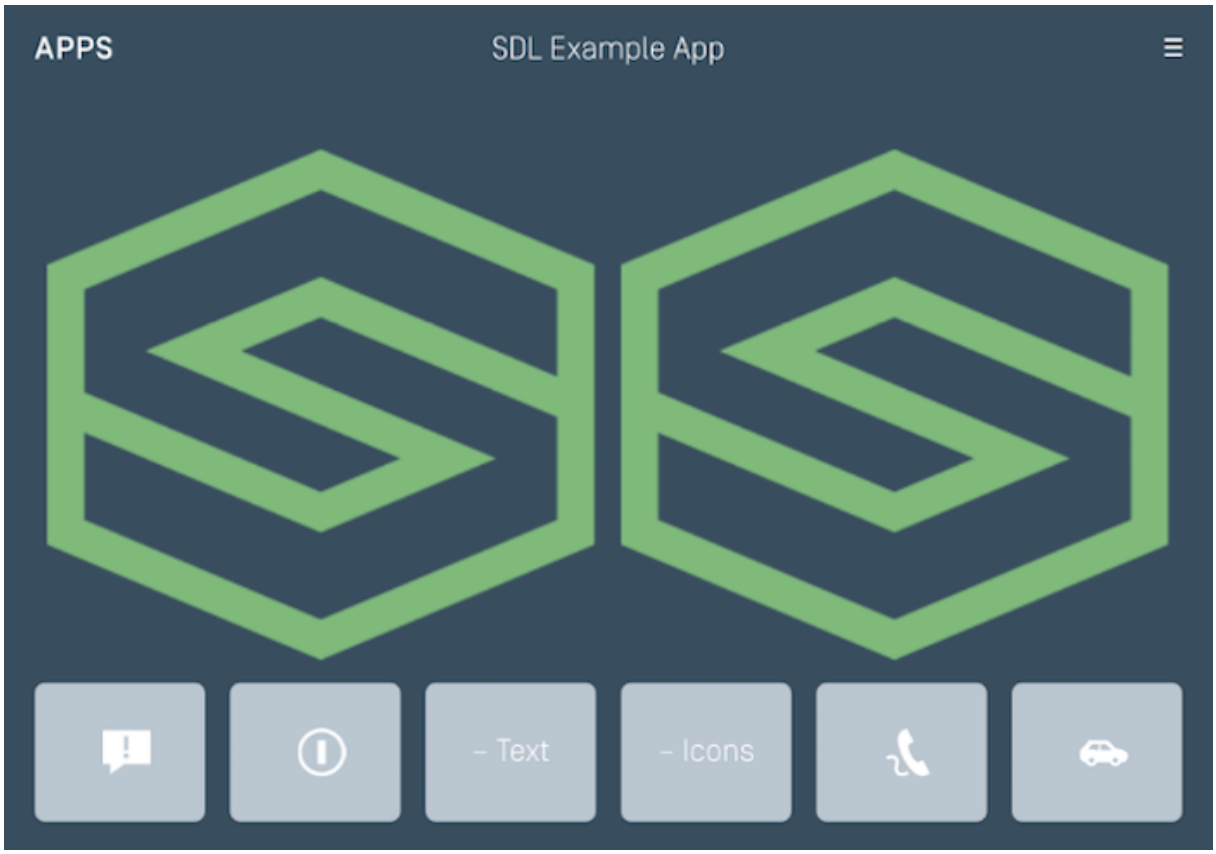


TEXT AND SOFT BUTTONS WITH GRAPHIC

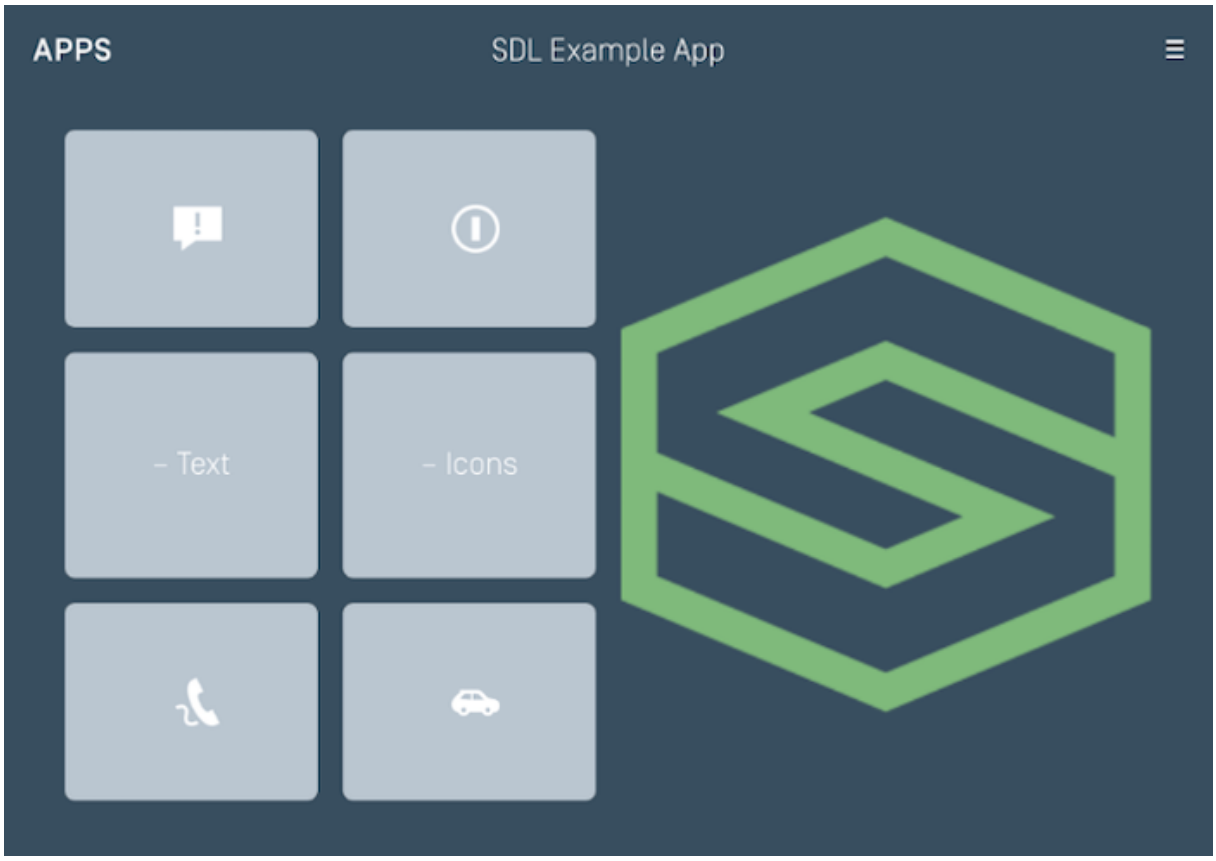


GRAPHIC WITH TEXT BUTTONS

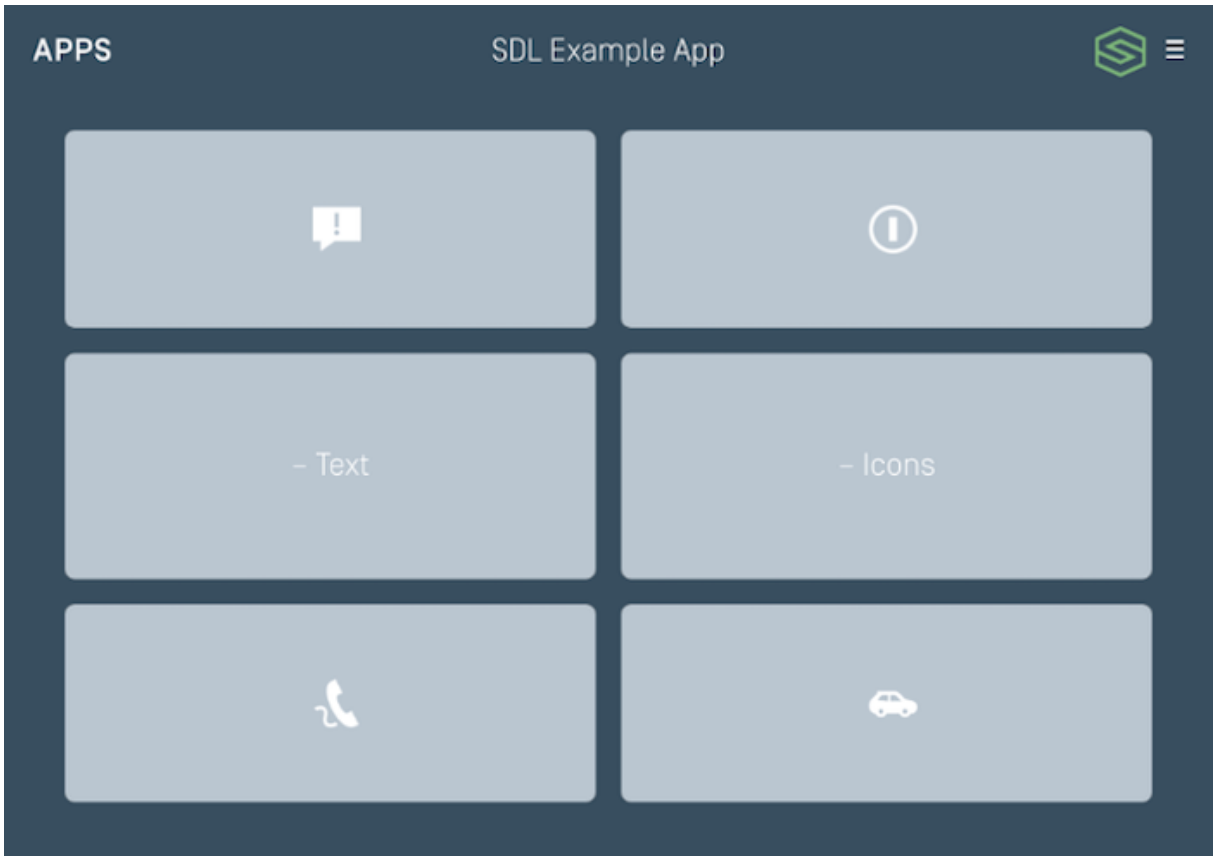
DOUBLE GRAPHIC WITH SOFT BUTTONS



TEXT BUTTONS WITH GRAPHIC



TEXT BUTTONS ONLY



LARGE GRAPHIC WITH SOFT BUTTONS

LARGE GRAPHIC ONLY

Template Text

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

Text Fields

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

Showing Text

```
sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setTextField1('Line 1 of Text');
sdlManager.getScreenManager().setTextField2('Line 2 of Text');
// Commit the updates and catch any errors
const success = await sdlManager.getScreenManager().commit().catch(function
(error) {
    // Handle Error
});
console.log('ScreenManager update complete:', success);
if (success === true) {
    // Update complete
} else {
    // Something went wrong
}
```

Removing Text

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

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

Template Images

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

Image Fields

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

Showing Images

Creating an SDLArtwork

Create an `SdlArtwork` object which can be manually uploaded or set into the `ScreenManager` and automatically uploaded. An `SdlArtwork` includes information about whether the image should be persisted between vehicle startups, whether the image is a template image and should be re-colored, and more.

```
const artwork = new SDL.manager.file.filetypes.SdlArtwork('artworkName', SDL.rpc.enums.FileType.GRAPHIC_PNG, fileData, true);
```

Setting Primary Graphic

```
sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setPrimaryGraphic(sdlArtwork);
// Commit the updates and catch any errors
const success = await sdlManager.getScreenManager().commit().catch(function
(error) {
    // Handle Error
});
console.log('ScreenManager update complete:', success);
if (success === true) {
    // Update complete
} else {
    // Something went wrong
}
```

Removing Images

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

```
sdlManager.getScreenManager().setPrimaryGraphic(null);
```

Overwriting Images

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

NOTE

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

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

Templating Images (RPC v5.0+)

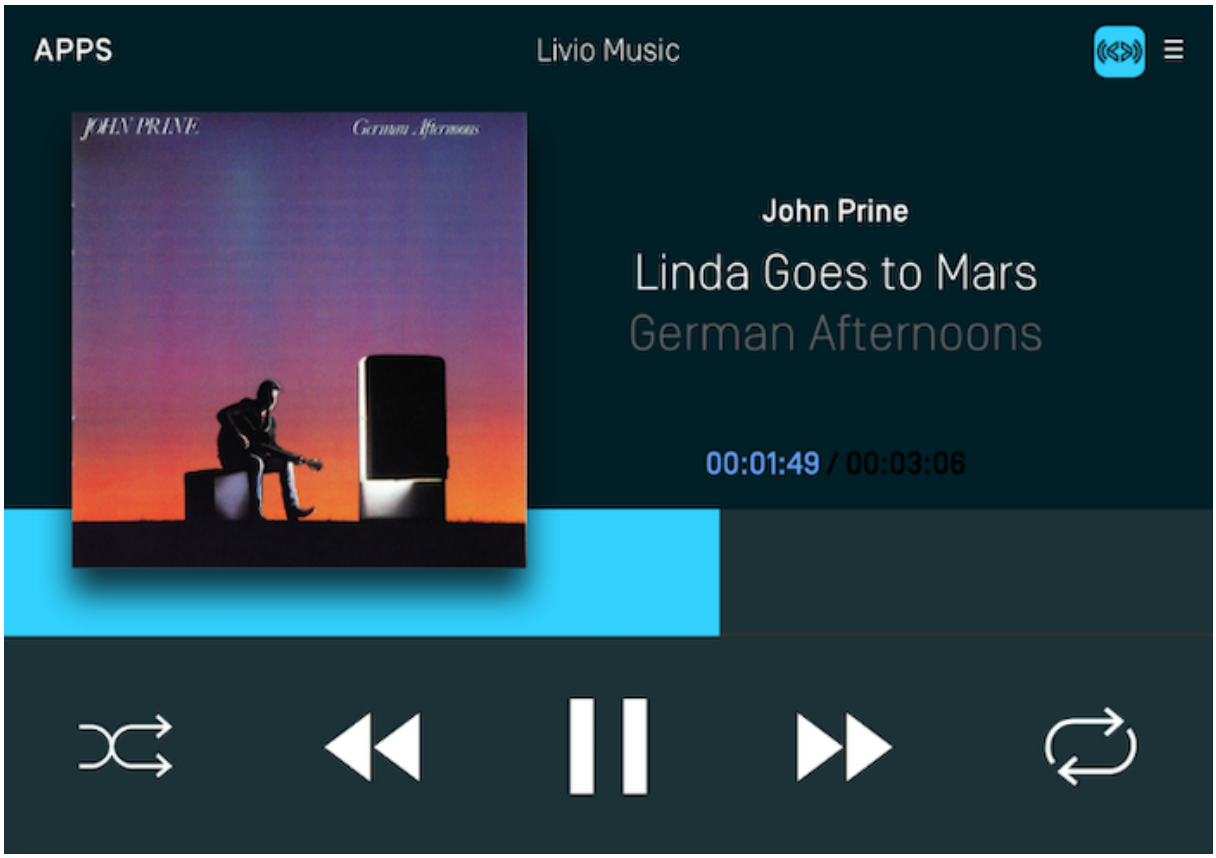
Templated images are tinted by Core so the image is visible regardless of whether your user has set the head unit to day or night mode. For example, if a head unit is in night mode with a dark theme (see Customizing the Template section for more details on how to customize theme colors), then your templated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

Soft buttons, menu icons, and primary / secondary graphics can all be templated. 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

```
const image = new SDL.manager.file.filetypes.SdlArtwork('artworkName',
SDL.rpc.enums.FileType.GRAPHIC_PNG, fileData, 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. Each OEM will design their own custom static icons but you can get an overview of the available icons from the icons designed for the open source Generic HMI. Static icons are fully supported by the screen manager via an `SdlArtwork` initializer. Static icons can be used in primary and secondary graphic fields, soft button image fields, and menu icon fields.

The SDL JavaScript Suite is currently missing support for the StaticIconName enum. This will be addressed in a future release. Constructing a StaticIcon can still be done using the appropriate hex values found here.

```
const staticIconAlbumName = '0x21';
const staticIconAlbumBytes = 0x21;
const staticIconArt = new SDL.manager.file.filetypes.SdlArtwork()
    .setName(staticIconAlbumName)
    .setFileData(staticIconAlbumBytes)
    .setStaticIcon(true)
    .setPersistent(false);
```

Template Custom Buttons

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

Soft Button Fields

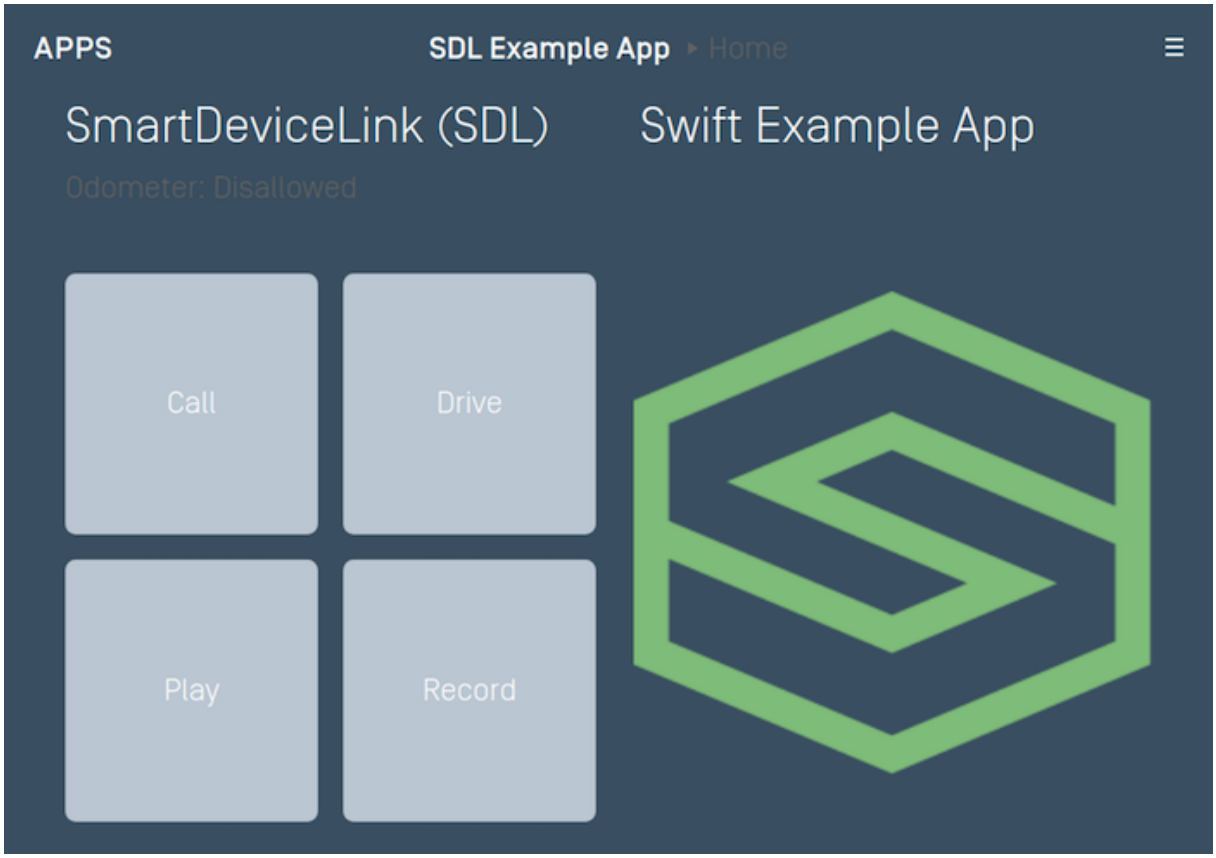
SCREENMANAGER PARAMETER NAME	DESCRIPTION
softButtonObjects	An array of buttons. Each template supports a different number of soft buttons

Creating Soft Buttons

To create a soft button using the `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 create all the states on initialization.

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

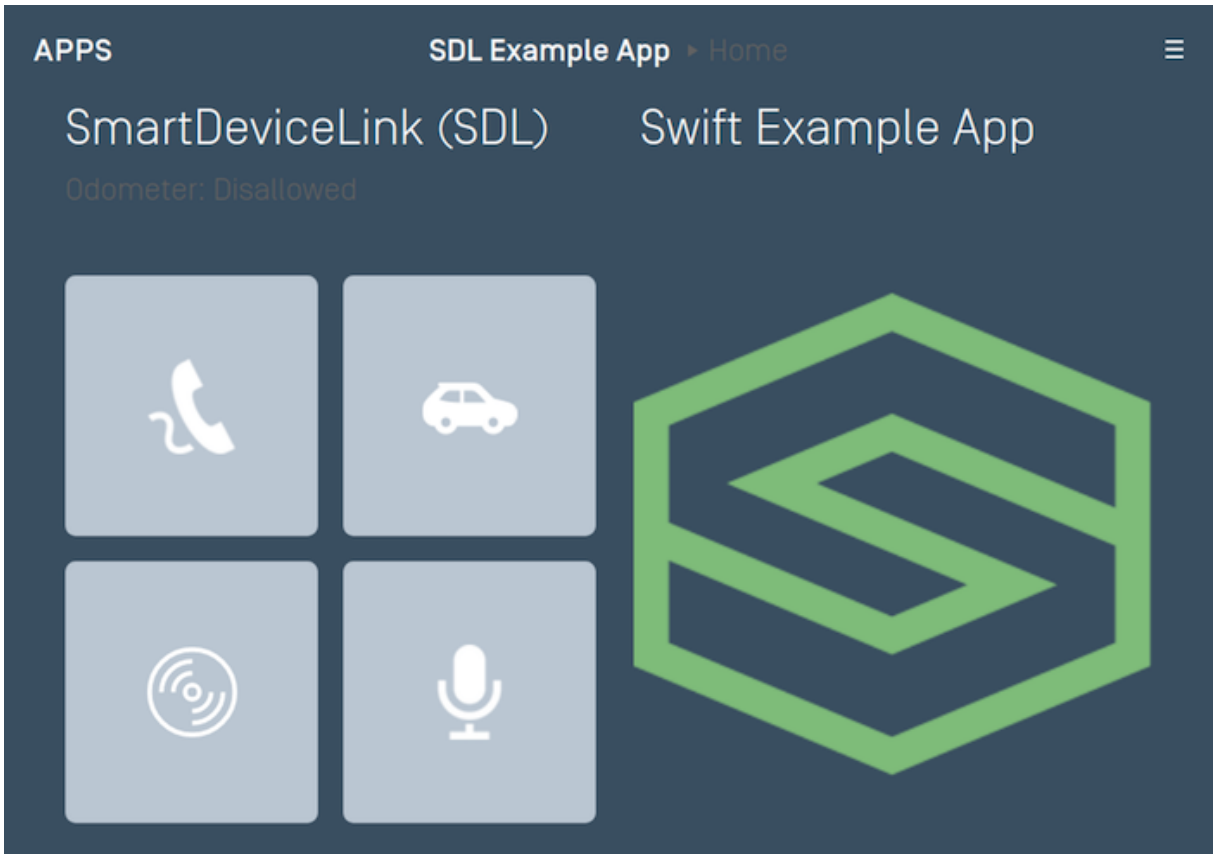
Text Only Soft Buttons



```
const textState = new new SDL.manager.screen.utils.SoftButtonState('State Name',
'Button Label Text');

const softButtonObject = new
SDL.manager.screen.utils.SoftButtonObject('softButtonObject', [textState],
textState.getName(), function (softButtonObject, rpc) {
    if (rpc instanceof SDL.rpc.messages.OnButtonPress) {
        console.log('SoftButton pressed!');
    }
});

sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setSoftButtonObjects([softButtonObject]);
// Commit the updates and catch any errors
const success = await sdlManager.getScreenManager().commit().catch(function
(error) {
    // Handle Error
});
console.log('ScreenManager update complete:', success);
if (success === true) {
    // Update complete
} else {
    // Something went wrong
}
```

Image Only Soft Buttons

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

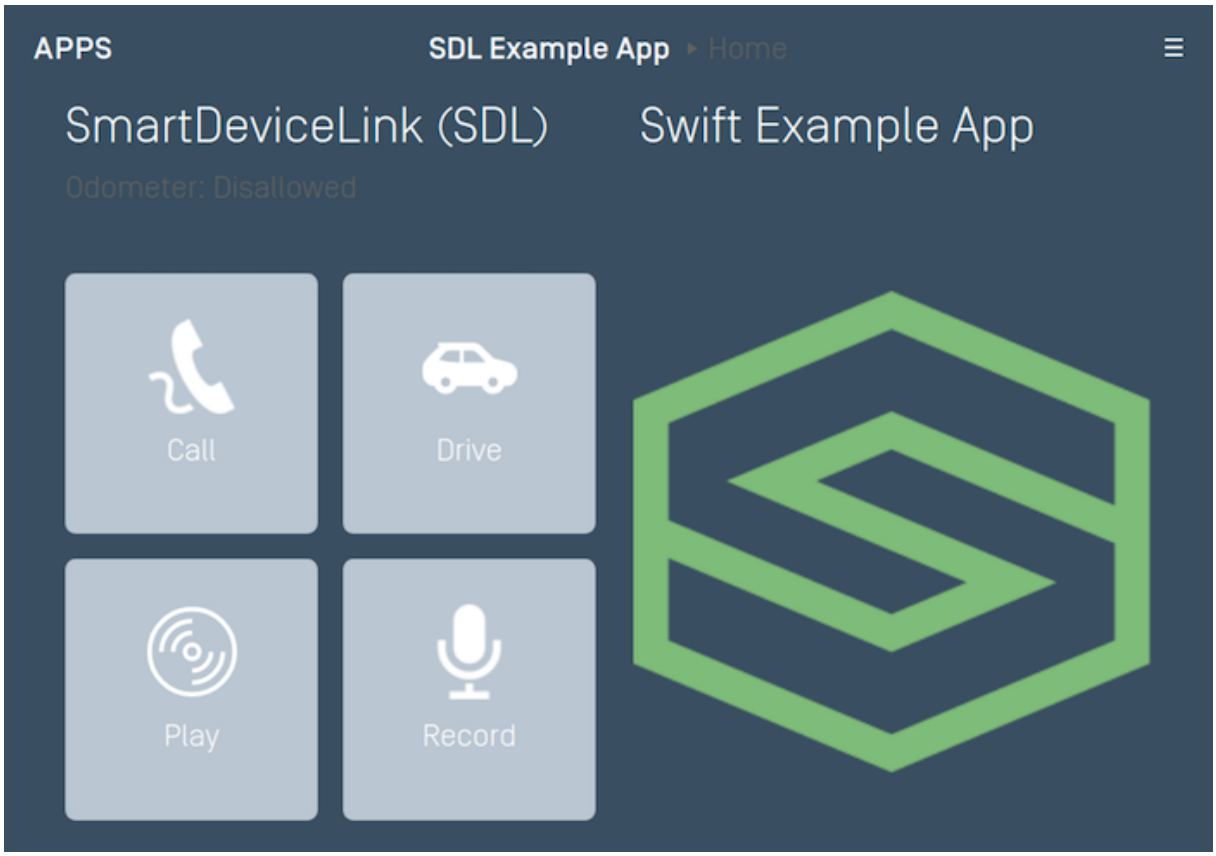



```
const softButtonCapabilitiesList =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().getSof

const imageSupported = (softButtonCapabilitiesList.length !== 0) ?
softButtonCapabilitiesList[0].getImageSupported() : false;
```

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

```
const imageState = new SDL.manager.screen.utils.SoftButtonState('State Name',
null, sdlArtwork);
const softButtonObject = new
SDL.manager.screen.utils.SoftButtonObject('softButtonObject', [imageState],
imageState.getName(), function (softButtonObject, rpc) {
    if (rpc instanceof SDL.rpc.messages.OnButtonPress) {
        console.log('SoftButton pressed');
    }
});

sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setSoftButtonObjects([softButtonObject]);
// Commit the updates and catch any errors
const success = await sdlManager.getScreenManager().commit().catch(function
(error) {
    // Handle Error
});
console.log('ScreenManager update complete:', success);
if (success === true) {
    // Update complete
} else {
    // Something went wrong
}
```

Image and Text Soft Buttons




```
const textAndImageState = new SDL.manager.screen.utils.SoftButtonState('State
Name', 'Button Label Text', artwork);
const softButtonObject = new
SDL.manager.screen.utils.SoftButtonObject('softButtonObject', [textAndImageState],
textAndImageState.getName(), function (softButtonObject, rpc) {
    if (rpc instanceof SDL.rpc.messages.OnButtonPress) {
        console.log('SoftButton pressed');
    }
});

sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setSoftButtonObjects([softButtonObject])
// Commit the updates and catch any errors
const success = await sdlManager.getScreenManager().commit().catch(function
(error) {
    // Handle Error
});
console.log('ScreenManager update complete:', success);
if (success === true) {
    // Update complete
} else {
    // Something went wrong
}
```

Highlighting a Soft Button

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

HIGHLIGHT ON

HIGHLIGHT OFF

```
const softButtonState1 = new SDL.manager.screen.utils.SoftButtonState('Soft
Button State Name', 'On', sdlArtwork);
softButtonState1.setHighlighted(true);
const softButtonState2 = new SDL.manager.screen.utils.SoftButtonState('Soft
Button State Name 2', 'Off', sdlArtwork);
softButtonState2.setHighlighted(false);
const softButtonObject = new
SDL.manager.screen.utils.SoftButtonObject('softButtonObject', [softButtonState1,
softButtonState2], softButtonState1.getName(), function (softButtonObj, rpc) {
    if (rpc instanceof SDL.rpc.messages.onButtonPress) {
        softButtonObject.transitionToNextState();
    }
});
```

Updating Soft Button States

When the soft button state needs to be updated, simply tell the `SoftButtonObject` to transition to the next state. If your button states do not cycle in a predictable order, you

can also tell the soft button which state to transition to by passing the `stateName` of the new soft button state.

```
const state1 = new SDL.manager.screen.utils.SoftButtonState('State1 Name',
'Button1 Label Text', sdlArtwork);
const state2 = new SDL.manager.screen.utils.SoftButtonState('State2 Name',
'Button2 Label Text', sdlArtwork);

const softButtonObject = new
SDL.manager.screen.utils.SoftButtonObject('softButtonObject', [state1, state2],
state1.getName(), function (softButtonObj, rpc) {
    if (rpc instanceof SDL.rpc.messages.OnButtonPress) {
        console.log('Soft Button pressed.');
    }
});

sdlManager.getScreenManager().beginTransaction();
sdlManager.getScreenManager().setSoftButtonObjects([]);
// Commit the updates and catch any errors
const success = await sdlManager.getScreenManager().commit().catch(function
(error) {
    // Handle Error
});
console.log('ScreenManager update complete:', success);
if (success === true) {
    // Update complete, transition to a new state
    const retrievedSoftButtonObject =
sdlManager.getScreenManager().getSoftButtonObjectByName('softButtonObject');
    retrievedSoftButtonObject.transitionToNextState();
} else {
    // Something went wrong
}
```

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([]);
```

Using RPCs

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

Template Subscription Buttons

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

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

Types of Subscription Buttons

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

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

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

Subscribing to Subscription Buttons

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

Subscribe with a Listener

Once you have subscribed to the button, the listener will be called when the button has been selected. If there is an error subscribing to the button the error message will be returned in the `catch` method.

```
const playPauseButtonListener = function (buttonName, buttonEvent) {
    if (onButton instanceof SDL.rpc.messages.OnButtonPress) {

    } else if (onButton instanceof SDL.rpc.messages.OnButtonEvent) {

    }
}

await sdlManager.getScreenManager()
    .addButtonListener(SDL.rpc.enums.ButtonName.PLAY_PAUSE,
playPauseButtonListener)
    .catch(function (err) {
        // Handle error
    });
```

Unsubscribing from Subscription Buttons

To unsubscribe to a subscription button, simply tell the `ScreenManager` which button name and listener object to unsubscribe.

```
await
sdlManager.getScreenManager().removeButtonListener(SDL.rpc.enums.ButtonName.
 playPauseButtonListener);
```

Media Buttons

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

subscribe to those buttons. However, those buttons will also trigger when the user uses the seek left / seek right buttons on the steering wheel.

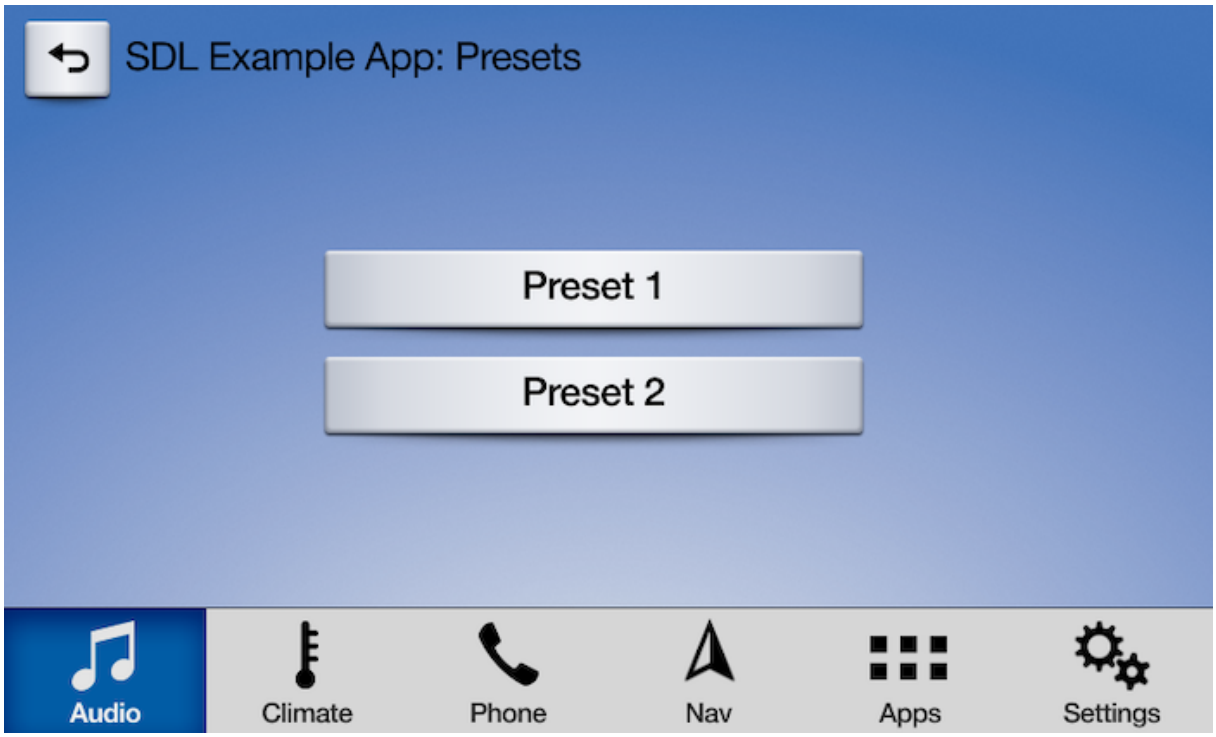
If desired, you can change the style of the play/pause button image between a play, stop, or pause icon by updating the audio streaming indicator, and you can also set the style of the next/previous buttons between a track or time seek style. See the Media Clock guide for more information.

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

```
await
sdlManager.getScreenManager().addButtonListener(SDL.rpc.enums.ButtonName.PLA
 function (buttonName, onButton) {
    if (onButton instanceof SDL.rpc.messages.OnButtonPress) {
        switch (onButton.getButtonPressMode()) {
            case SDL.rpc.enums.ButtonPressMode.SHORT:
                // The user short pressed the button
            case SDL.rpc.enums.ButtonPressMode.LONG:
                // The user long pressed the button
        }
    } else if (onButton instanceof SDL.rpc.messages.OnButtonEvent) {
        // OnButtonEvent
    }
}).catch(function (info) {
    // There was an error subscribing to the button
});
```

Preset Buttons

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



Checking if Preset Buttons are Supported

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

```
const numOfCustomPresetsAvailable =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().getNu
```

Subscribing to Preset Buttons

```
function onButtonListener (buttonName, onButton) {
    if (onButton instanceof SDL.rpc.messages.OnButtonPress) {
        switch (buttonName) {
            case SDL.rpc.enums.ButtonName.PRESET_1:
                // The user short or long pressed the preset 1 button
            case SDL.rpc.enums.ButtonName.PRESET_2:
                // The user short or long pressed the preset 2 button
        }
    } else if (onButton instanceof SDL.rpc.messages.OnButtonEvent) {
        // OnButtonEvent
    }
}

function onError (info) {
    // There was an error subscribing to the button
}

sdlManager.getScreenManager().addButtonListener(SDL.rpc.enums.ButtonName.PRE
 onButtonListener).catch(onError);
sdlManager.getScreenManager().addButtonListener(SDL.rpc.enums.ButtonName.PRE
 onButtonListener).catch(onError);
```

Navigation Buttons

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

own from your map screen. If subscribing to these buttons fails, you can display buttons of your own on your map screen.

Subscribing to Navigation Buttons

```
await
sdlManager.getScreenManager().addButtonListener(SDL.rpc.enums.ButtonName.NAV
 function (buttonName, onButton) {
    if (onButton instanceof SDL.rpc.messages.OnButtonPress) {
        switch (onButton.getButtonPressMode()) {
            case SDL.rpc.enums.ButtonPressMode.SHORT:
                // The user short pressed the button
            case SDL.rpc.enums.ButtonPressMode.LONG:
                // The user long pressed the button
        }
    } else if (onButton instanceof SDL.rpc.messages.OnButtonEvent) {
        // OnButtonEvent
    }
}).catch(function (info) {
    // There was an error subscribing to the button
});
```

Main Menu

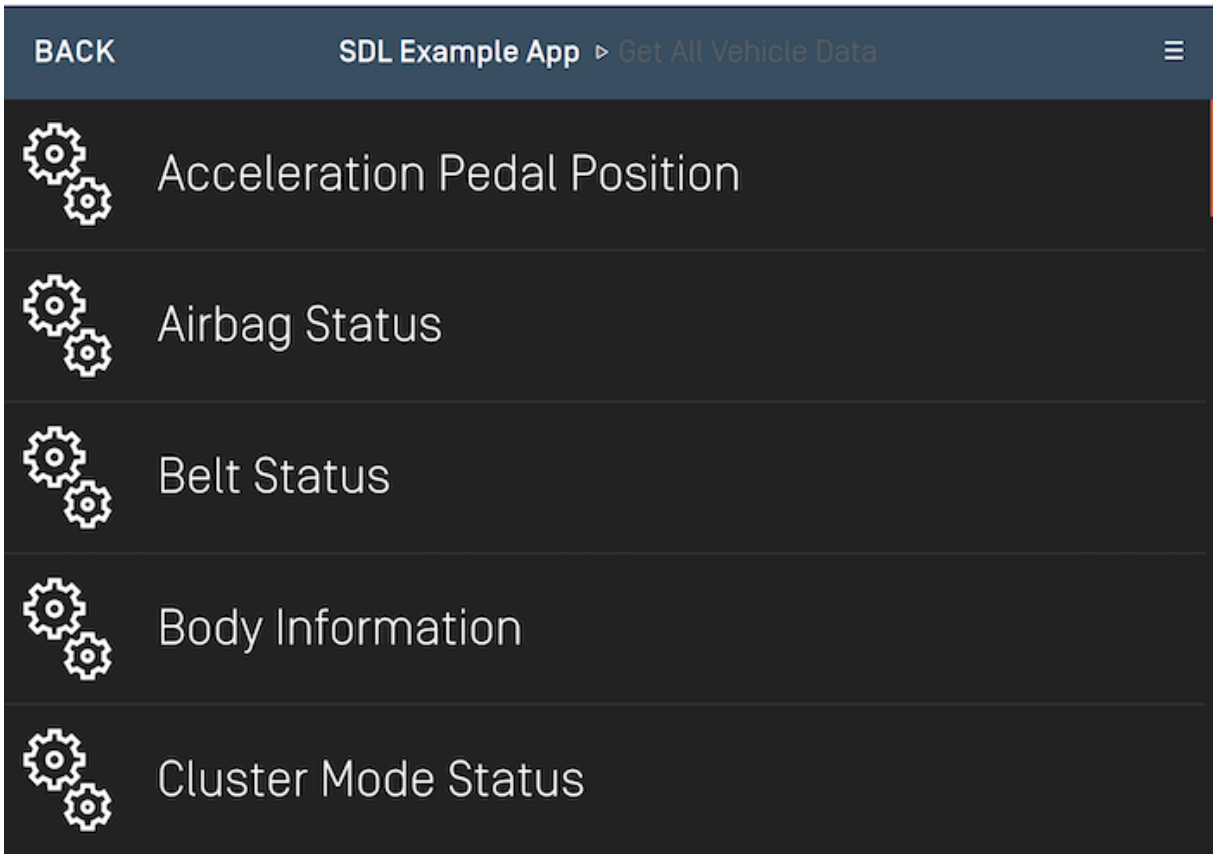
You have two different options when creating menus. One is to simply add items to the default menu available in every template. The other is to create a custom menu that pops up when needed. You can find more information about these popups in the Popup Menus section. This guide will cover using the default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

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

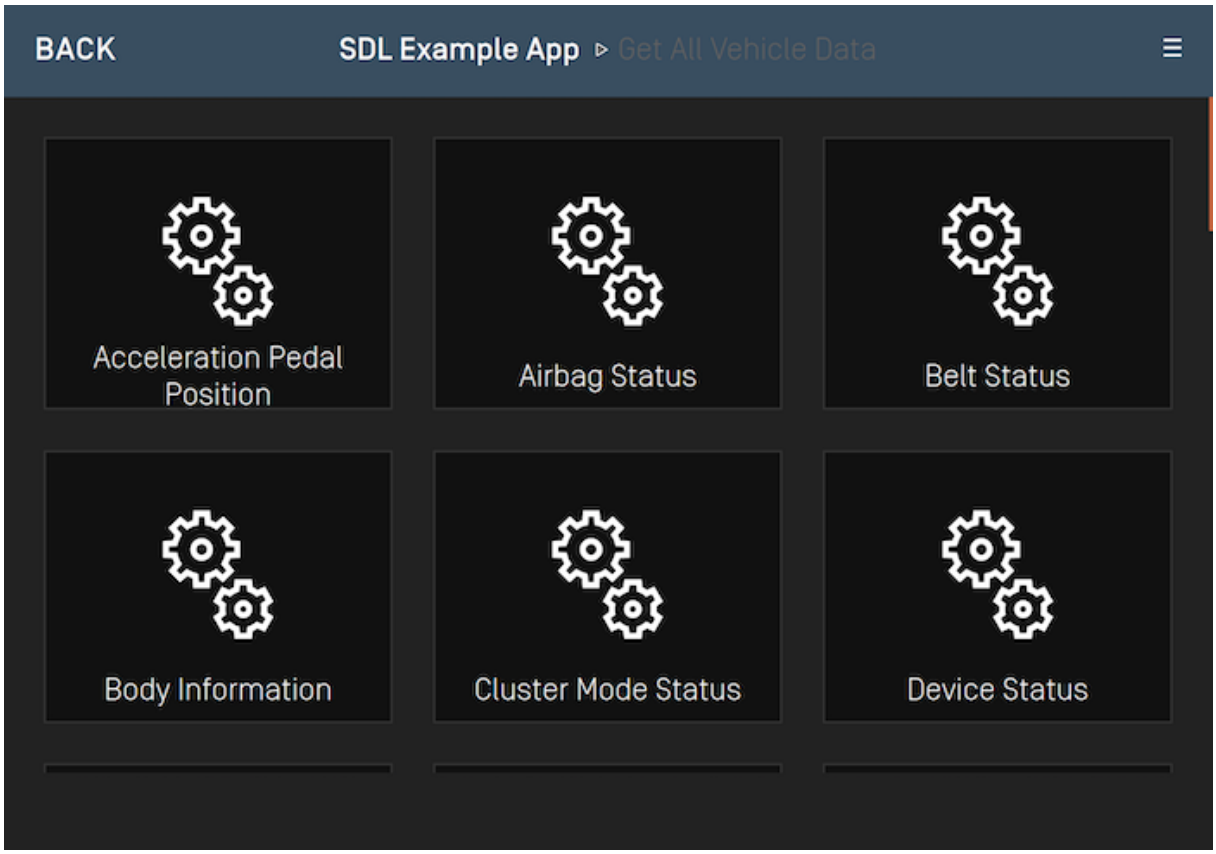
Setting the Menu Layout (RPC v6.0+)

On some newer head units, you may have the option to display menu items as a grid of tiles instead of the default list layout. To determine if the head unit supports the tiles layout, check the `SystemCapabilityManager`'s `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.

LIST MENU LAYOUT



GRID MENU LAYOUT

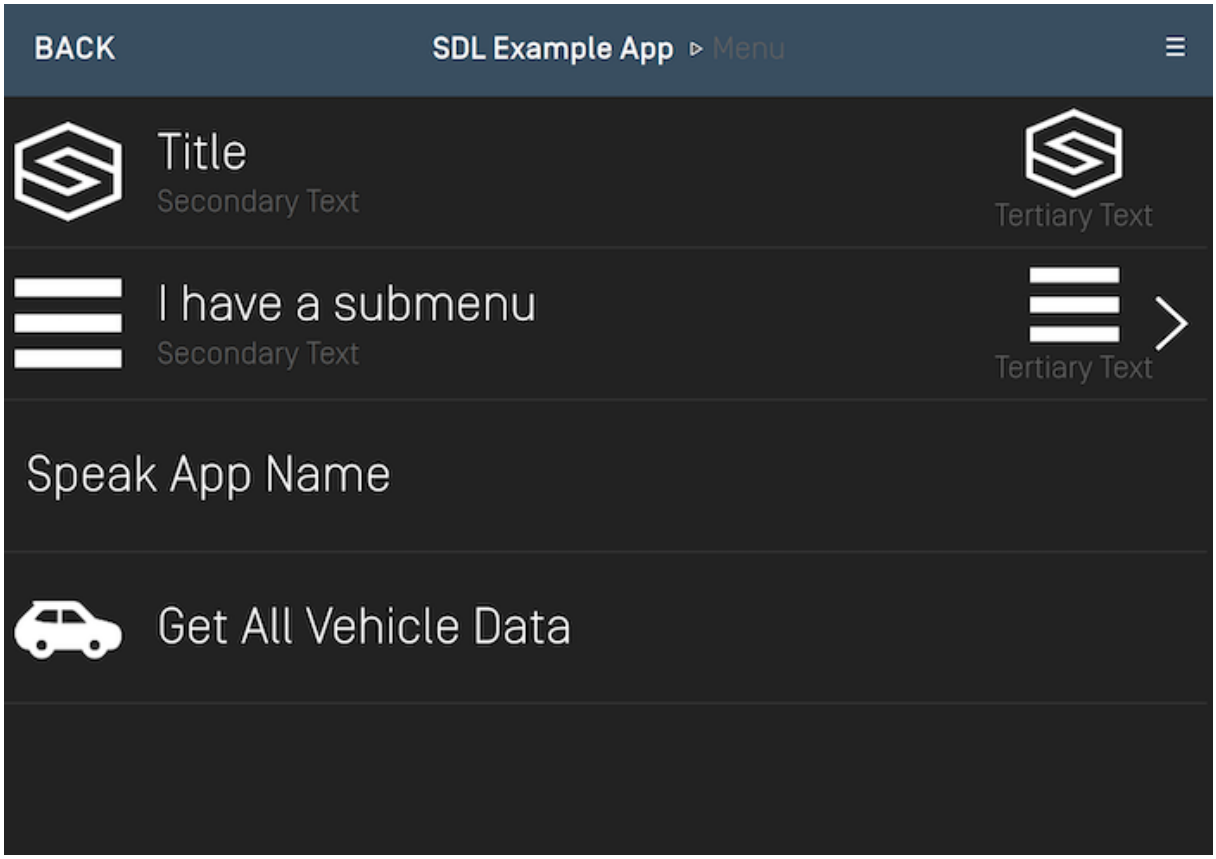



```
const menuConfiguration = new SDL.manager.screen.menu.MenuConfiguration()
    .setMenuLayout(mainMenuLayout)
    .setSubMenuLayout(submenuLayout);
sdlManager.getScreenManager().setMenuConfiguration(menuConfiguration);
```

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.



NOTE

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

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

```
// Create the menu cell
const cell = new SDL.manager.screen.menu.MenuCell('Cell Text')
    .setSecondaryText('Secondary Text')
    .setTertiaryText('Tertiary Text')
    .setVoiceCommands(['cell text'])
    .setMenuSelectionListener(new
SDL.manager.screen.menu.MenuSelectionListener()
        .setOnTriggered((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
        }));
```

Adding Submenus

Adding a submenu is as simple as adding subcells to a `MenuCell`. 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
const innerCell = new SDL.manager.screen.menu.MenuCell('inner menu cell')
    .setSecondaryText('secondary text')
    .setTertiaryText('tertiary text')
    .setVoiceCommands(['inner menu cell'])
    .setMenuSelectionListener(new
SDL.manager.screen.menu.MenuSelectionListener()
        .setOnTriggered((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
const cell = new SDL.manager.screen.menu.MenuCell('cell')
    .setSecondaryText('secondary text')
    .setTertiaryText('tertiary text')
    .setSubMenuLayout(SDL.rpc.enums.MenuLayout.LIST)
    .setSubCells([innerCell]);

sdlManager.getScreenManager().setMenu([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.

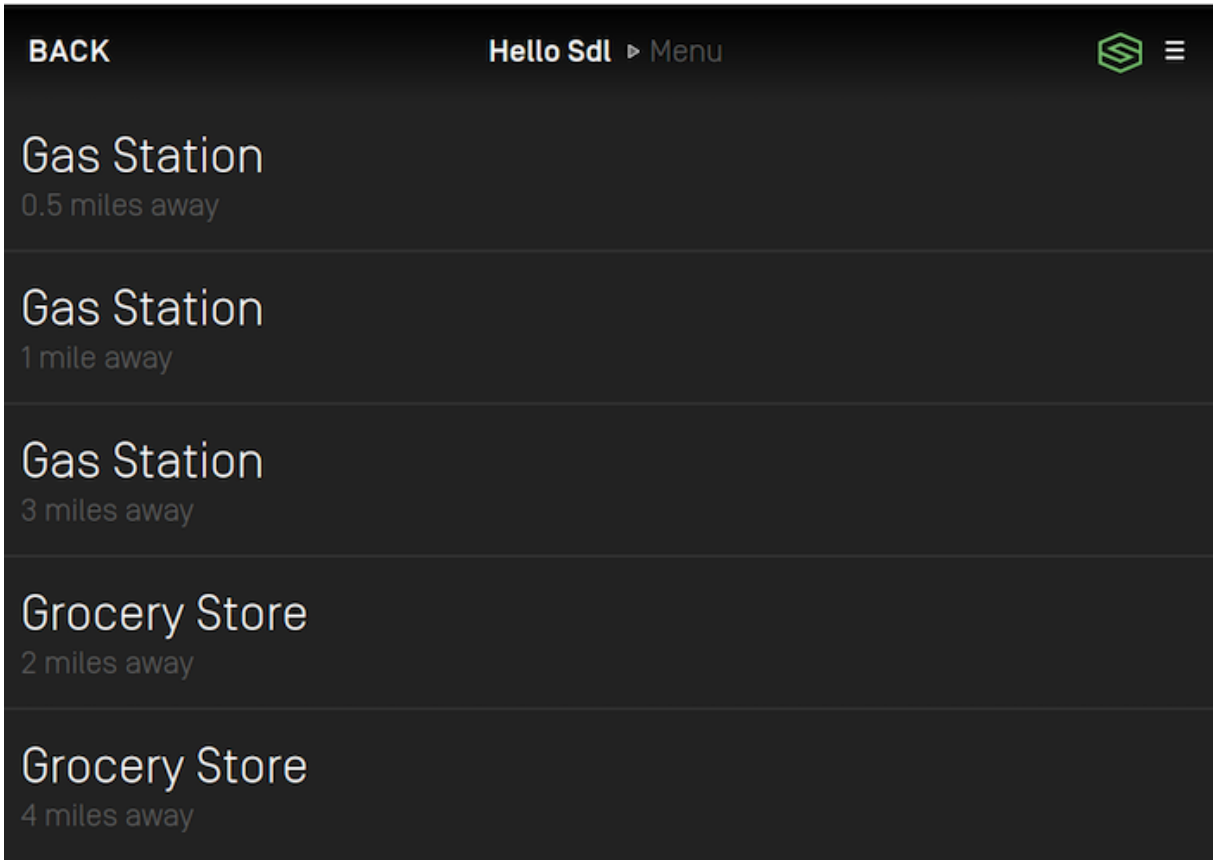
If you are doing this manually, you must use the `DeleteCommand` and `DeleteSubMenu` RPCs, passing the `cmdID`s you wish to delete.

Duplicate Menu Titles

Starting with SDL v1.5+ menu cells and sub-menu cells do not require unique titles in order to be presented. For example, if you are trying to display points of interest as a list you can now have multiple locations with the same name but are not the same location. You cannot present multiple cells that are exactly the same. They must have some property that makes them different, such as `secondaryText` or an artwork.

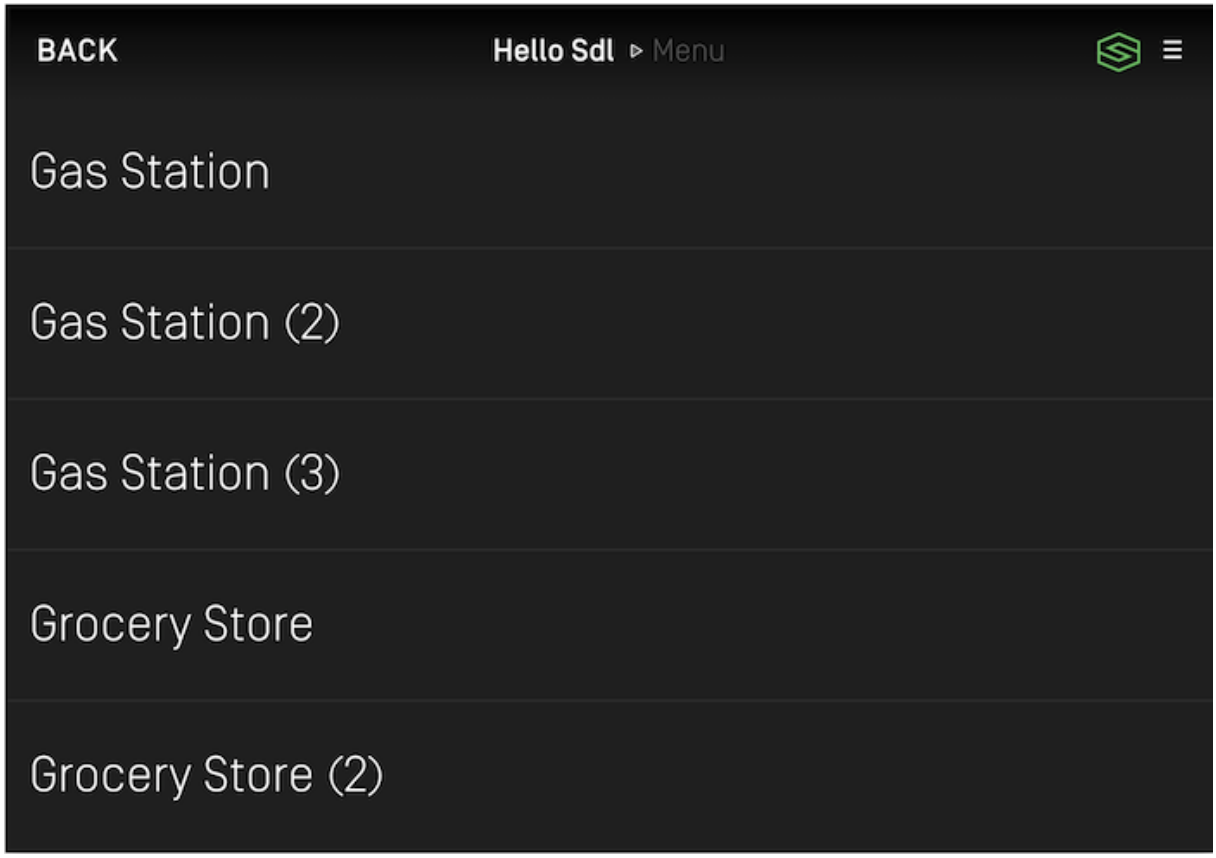
RPC V7.1+ CONNECTIONS

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



RPC V7.0 AND BELOW CONNECTIONS

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



Using RPCs

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

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

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

Popup Menus

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

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

Presenting a Popup Menu

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

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

Creating Cells

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.

```
const cell = new SDL.manager.screen.choiceset.ChoiceCell("cell1 text")
    .setVoiceCommands(["cell1"])
const fullCell = new SDL.manager.screen.choiceset.ChoiceCell("cell2 text")
    .setSecondaryText("cell2 secondaryText")
    .setTertiaryText("cell2 tertiaryText")
    .setVoiceCommands(["cell2"])
    .setArtwork(image1Artwork)
    .setSecondaryArtwork(image2Artwork)
```

Preloading Cells

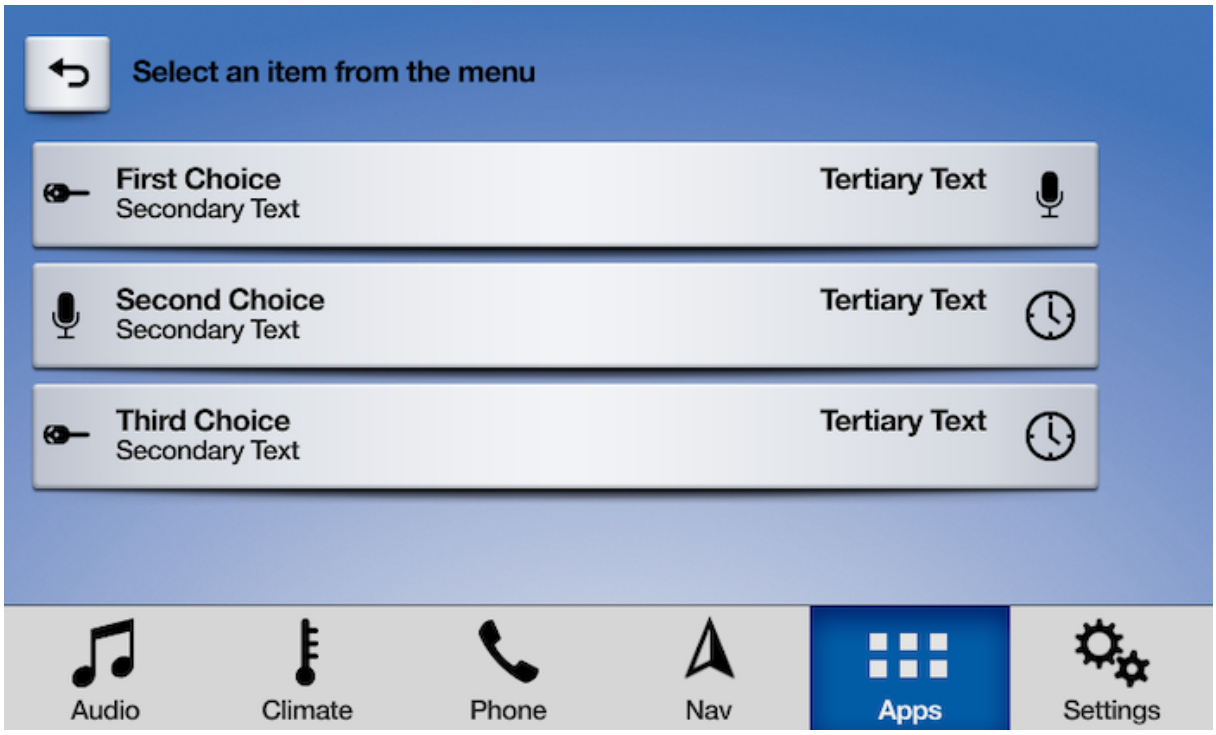
If you know the content you will show in the popup menu long before the menu is shown to the user, you can "preload" those cells in order to speed up the popup menu presentation at a later time. Once you preload a cell, you can reuse it in multiple popup menus without having to send the cell content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

```
const success = await sdlManager.getScreenManager().preloadChoices([cell, fullCell]);
```

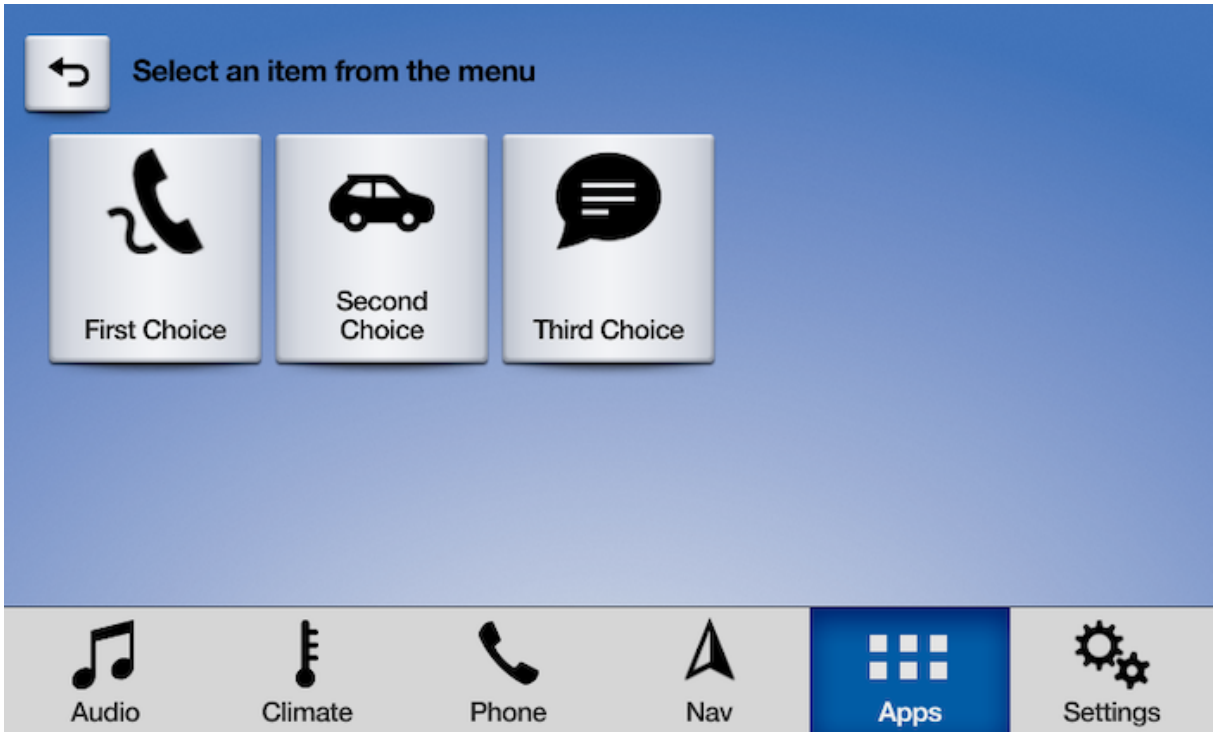
Presenting a Menu

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
Audio
Climate
Phone
Nav
Apps
Settings

MENU - ICON


Select an item from the menu
First Choice
Second Choice
Third Choice
Audio
Climate
Phone
Nav
Apps
Settings

> **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 or a list. If you are using tiles, it's recommended to use artworks on each item.

```
const listener = new SDL.manager.screen.choiceset.ChoiceSetSelectionListener()
    .setOnChoiceSelected((choiceCell, triggerSource, 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
    })
    .setOnError((error) => {
        // handle error
    });
const choiceSet = new SDL.manager.screen.choiceset.ChoiceSet("ChoiceSet Title",
[cell, fullCell], listener);
```

PRESENTING THE MENU WITH A MODE

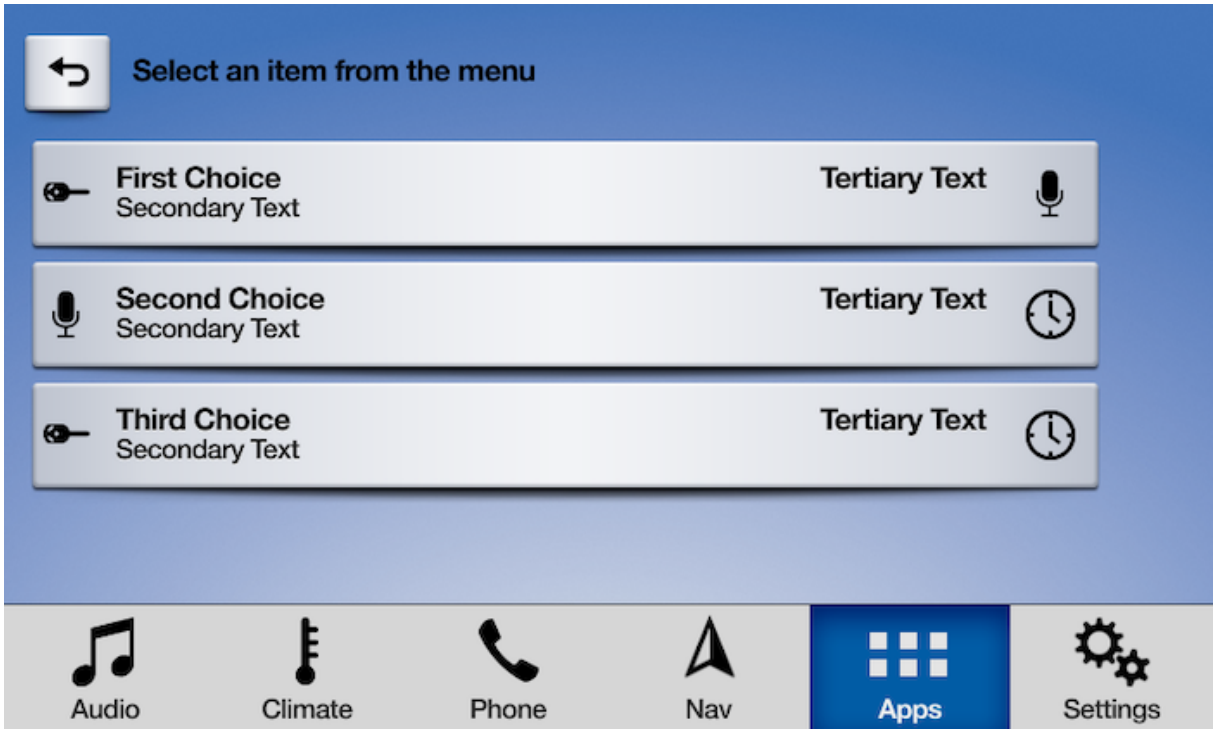
Finally, you will present the menu. When you do so, you must choose a `mode` to present it in. If you have no `vrCommands` on the choice cell you should choose `manualOnly`. If `vrCommands` are available, you may choose `voiceRecognitionOnly` or `both`.

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

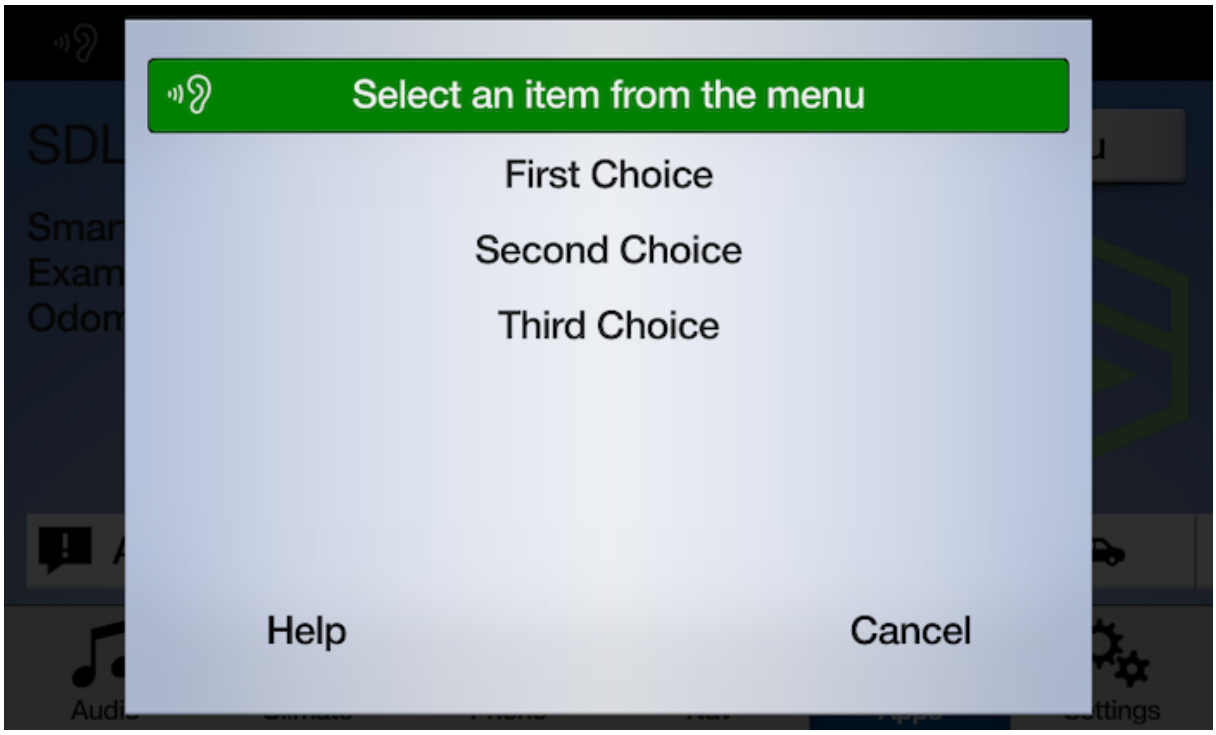
It may seem that the answer is to always use `both`. However, remember that you must provide `vrCommand`s on all cells to use `both`, which is exponentially slower than not providing `vrCommand`s (this is especially relevant for large menus, but less important for smaller ones). Also, some head units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

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

MENU - MANUAL ONLY MODE



MENU - VOICE ONLY MODE

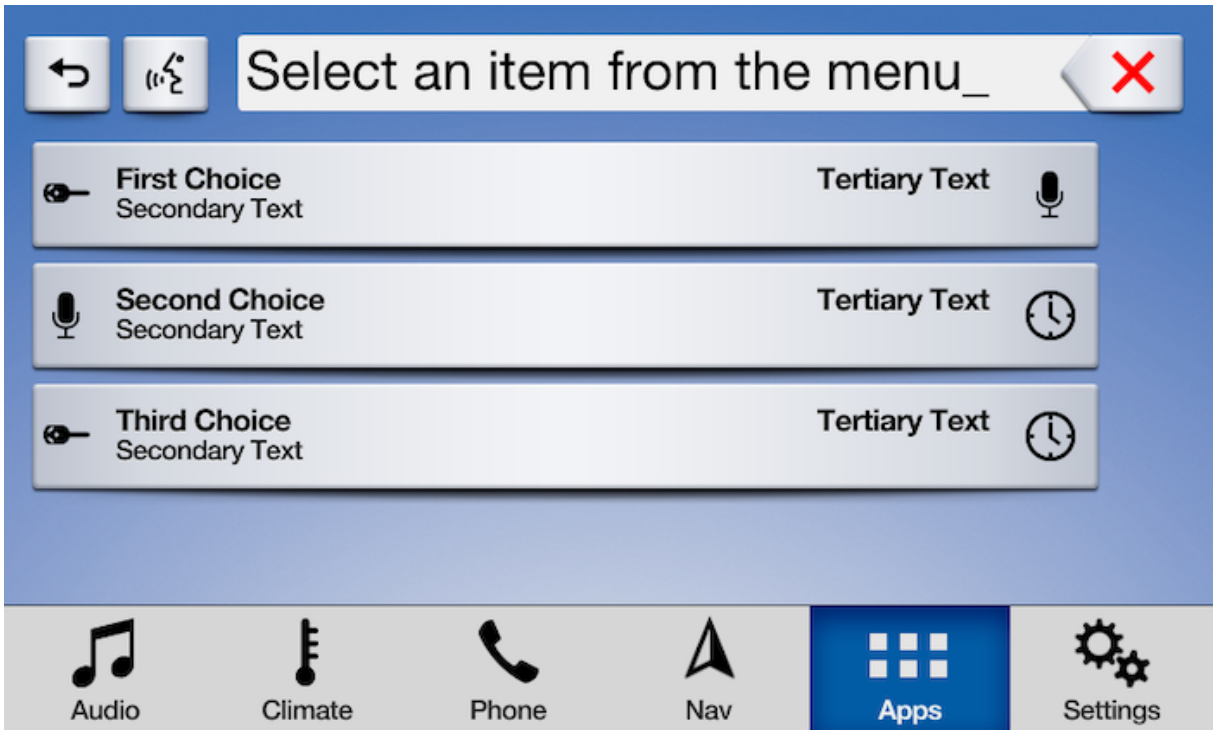



```
sdlManager.getScreenManager().presentChoiceSet(choiceSet,
SDL.rpc.enums.InteractionMode.MANUAL_ONLY);
```

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,
SDL.rpc.enums.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.

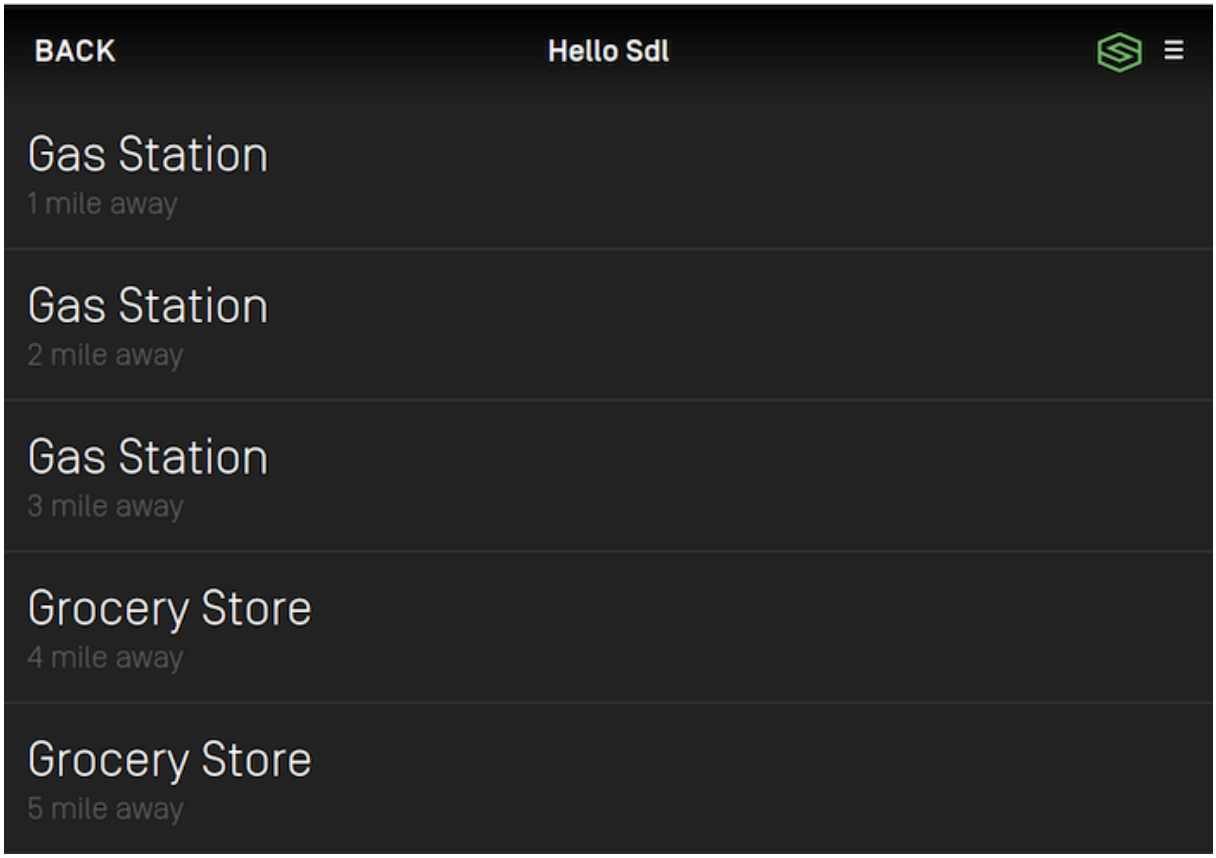
```
choiceSet.cancel();
```

Duplicate Cell Titles

Starting with SDL v1.3.0+ choice cells no longer require unique titles in order to be presented. For example, if you are trying to display points of interest as a list you can now have multiple locations with the same name but are not the same location. You cannot present multiple cells that are exactly the same. They must have some property that makes them different, such as `secondaryText` or an artwork.

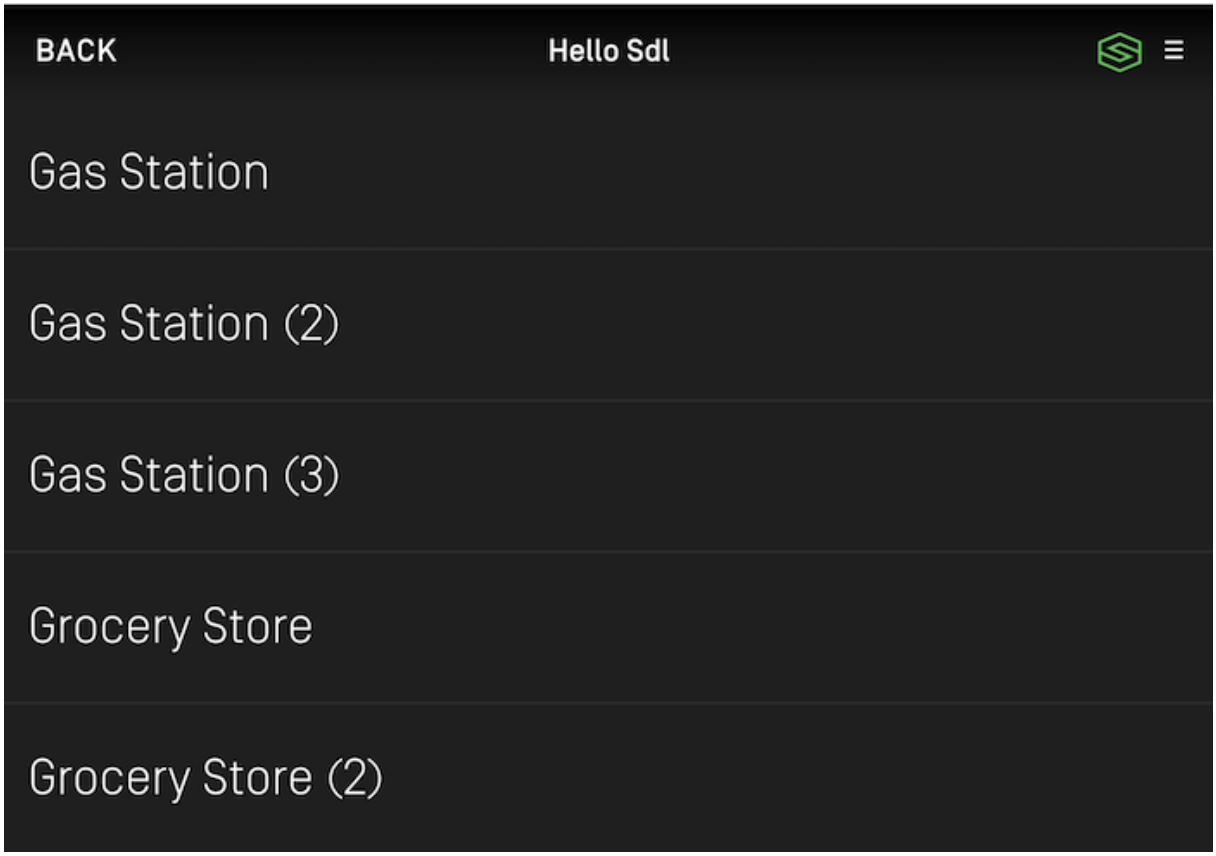
RPC V7.1+ CONNECTIONS

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



RPC V7.0 AND BELOW CONNECTIONS

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



Using RPCs

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

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

Popup Keyboards

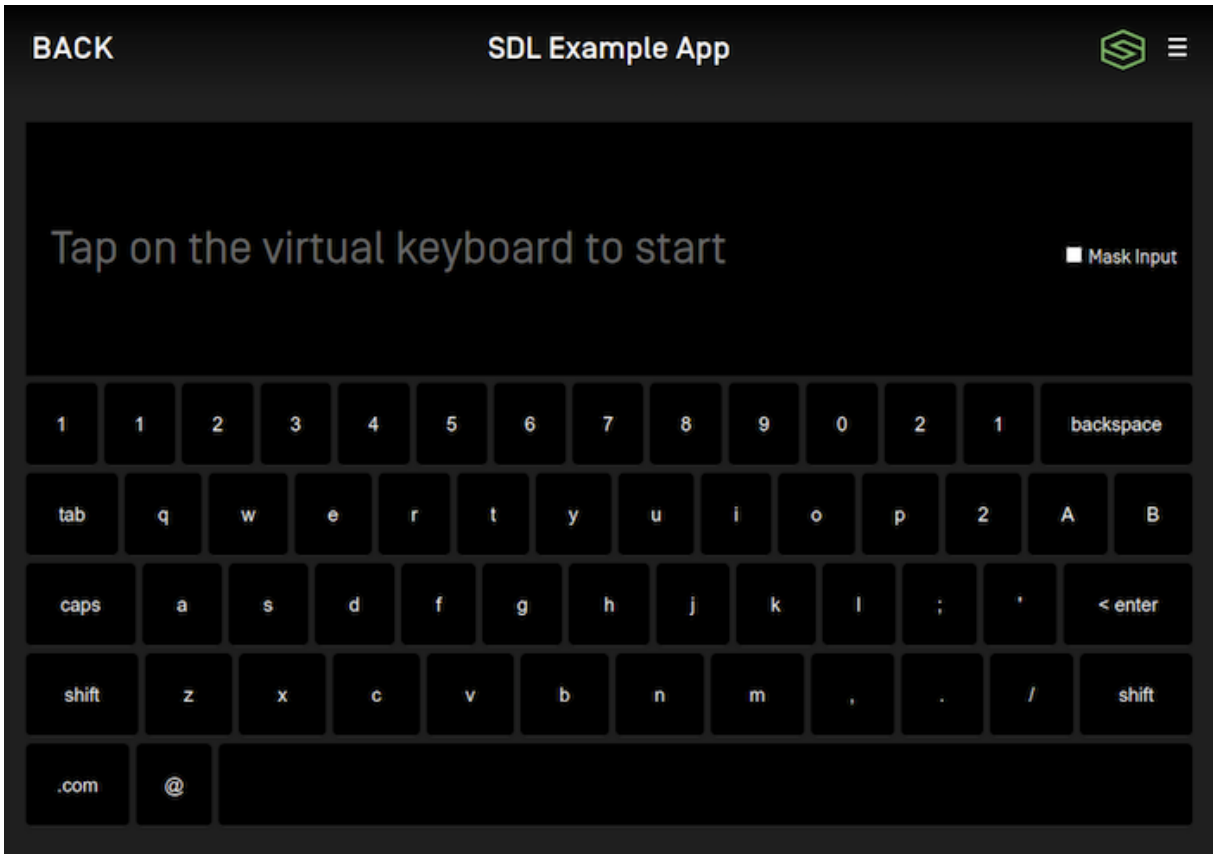
Presenting a keyboard or a popup menu with a search field requires you to implement the `KeyboardListener`. Note that the `initialText` in the keyboard case often acts as "placeholder text" and not as true initial text.

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.




```
const cancelId = sdlManager.getScreenManager().presentKeyboard('Initial text', null, keyboardListener);
```

Implementing the Keyboard Listener

Using the `KeyboardListener` involves implementing several methods:

```
const keyboardListener = new SDL.manager.screen.choiceset.KeyboardListener()
    .setOnUserDidSubmitInput((inputText, event) => {
        switch (event) {
            case SDL.rpc.enums.KeyboardEvent.ENTRY_VOICE:
                // The user decided to start voice input, you should start an AudioPassThru
session if supported
                break;
            case SDL.rpc.enums.KeyboardEvent.ENTRY_SUBMITTED:
                // The user submitted some text with the keyboard
                break;
            default:
                break;
        }
    })
    .setOnKeyboardDidAbortWithReason((event) => {
        switch (event) {
            case SDL.rpc.enums.KeyboardEvent.ENTRY_CANCELLED:
                // The user cancelled the keyboard interaction
                break;
            case SDL.rpc.enums.KeyboardEvent.ENTRY_ABORTED:
                // The system aborted the keyboard interaction
                break;
            default:
                break;
        }
    })
    .setUpdateAutocompleteWithInput((currentInputText,
keyboardAutocompleteCompletionListener) => {
        // Check the input text and return a list of autocomplete results
        keyboardAutocompleteCompletionListener(updatedAutoCompleteList);
    })
    .setUpdateCharacterSetWithInput((currentInputText,
keyboardCharacterSetCompletionListener) => {
        // Check the input text and return a set of characters to allow the user to enter
    })
    .setOnKeyboardDidSendEvent((event, currentInputText) => {
        // This is sent upon every event, such as keypresses, cancellations, and aborting
    })
    .setOnKeyboardDidUpdateInputMask((event) => {
        switch (event) {
            case SDL.rpc.enums.KeyboardEvent.INPUT_KEY_MASK_ENABLED:
                // The user enabled input key masking
                break;
            case SDL.rpc.enums.KeyboardEvent.INPUT_KEY_MASK_DISABLED:
                // The user disabled input key masking
                break;
            default:
                break;
        }
    });
```

Configuring Keyboard Properties

You can change default keyboard properties by updating `sdlManager.getScreenManager().setKeyboardConfiguration()`. If you want to change the keyboard configuration for only one keyboard session and keep the default keyboard configuration unchanged, you can pass a single-use `KeyboardProperties` to `presentKeyboard()`.

KEYBOARD LANGUAGE

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

```
const keyboardConfiguration = new SDL.rpc.structs.KeyboardProperties()
    .setLanguage(SDL.rpc.enums.Language.EN_US);

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

LIMITED CHARACTER LIST

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

```
const keyboardConfiguration = new SDL.rpc.structs.KeyboardProperties()
    .setLimitedCharacterList(['a', 'b', 'c']);

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

AUTOCOMPLETE LIST

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

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

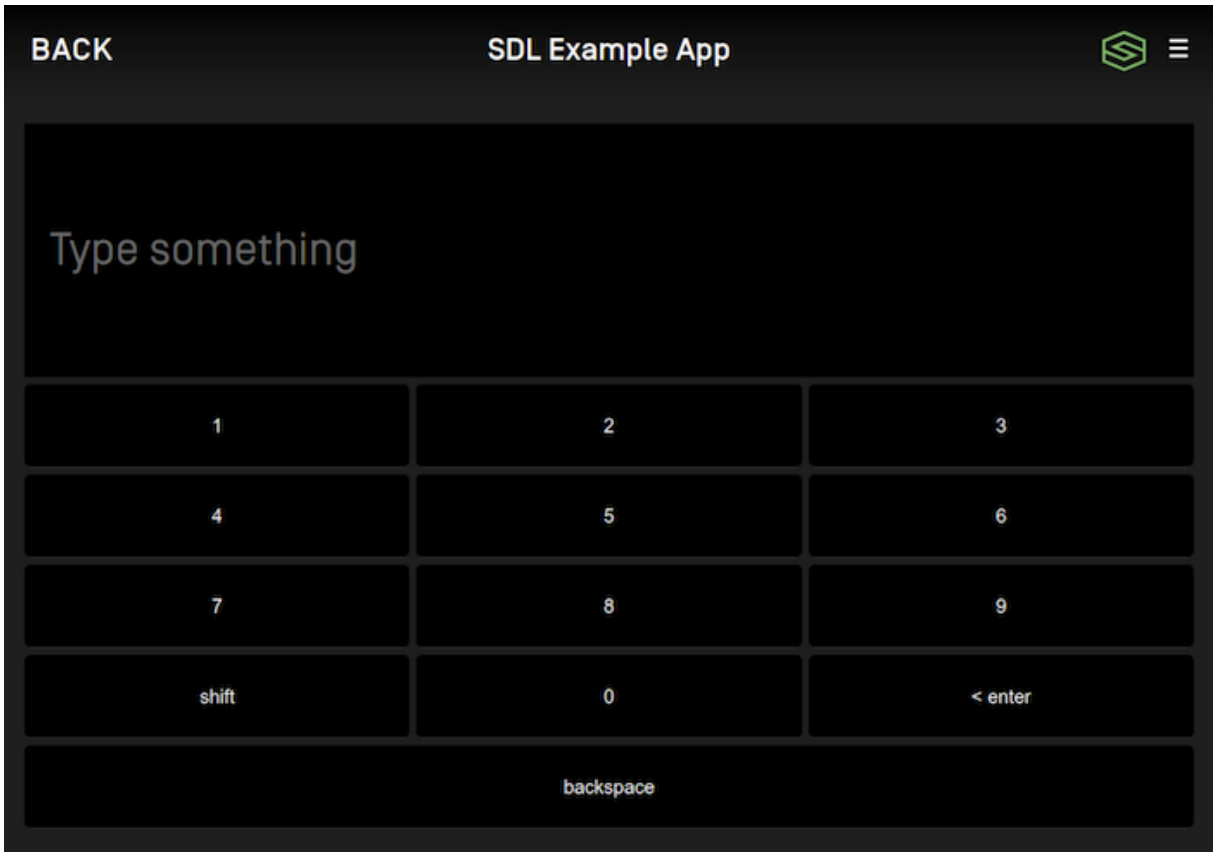
```
const keyboardConfiguration = new SDL.rpc.structs.KeyboardProperties()
    .setAutoCompleteList(['test1', 'test2', 'test3']);

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

KEYBOARD LAYOUT

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

NOTE

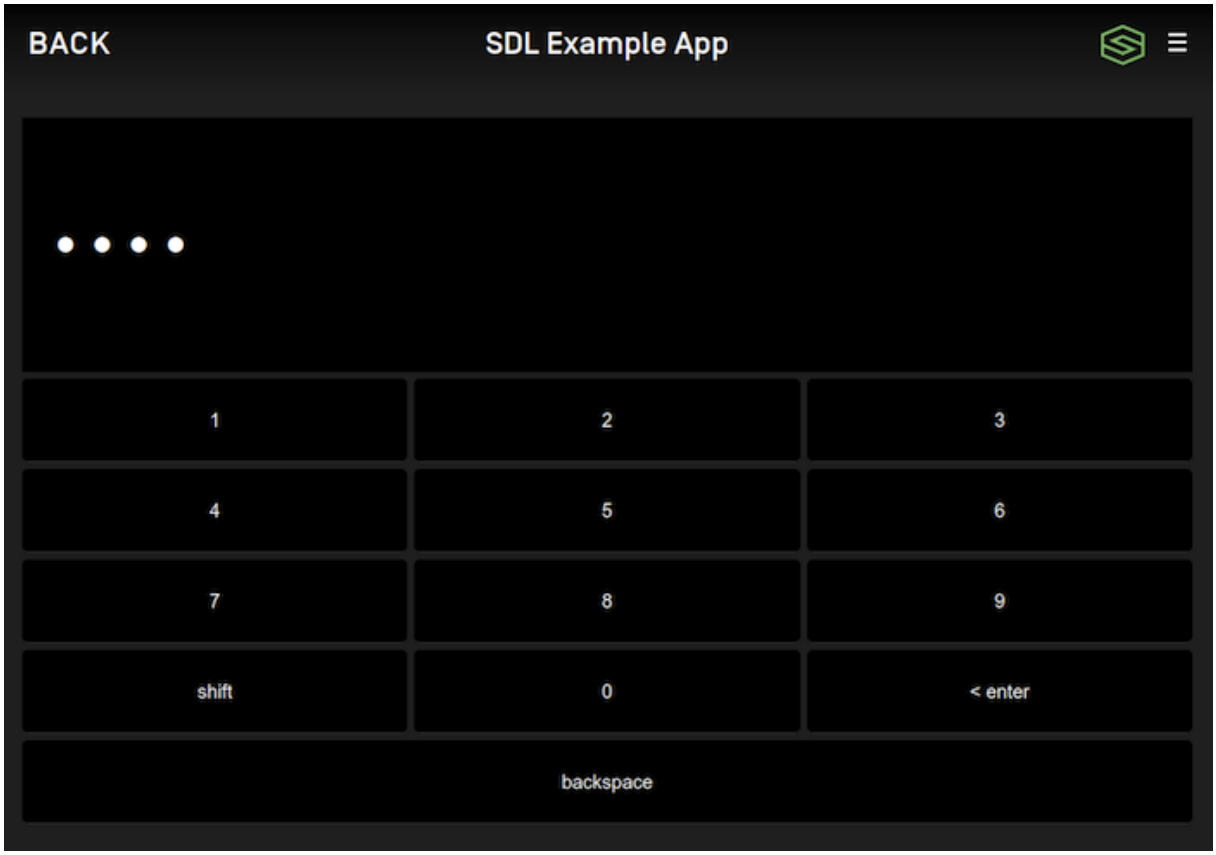
The numeric keyboard layout is only available on RPC 7.1+. See the section Checking Keyboard Capabilities to determine if this layout is available.




```
const keyboardConfiguration = new SDL.rpc.structs.KeyboardProperties()
    .setKeyboardLayout(SDL.rpc.enums.KeyboardLayout.NUMERIC);

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

INPUT MASKING (RPC 7.1+)

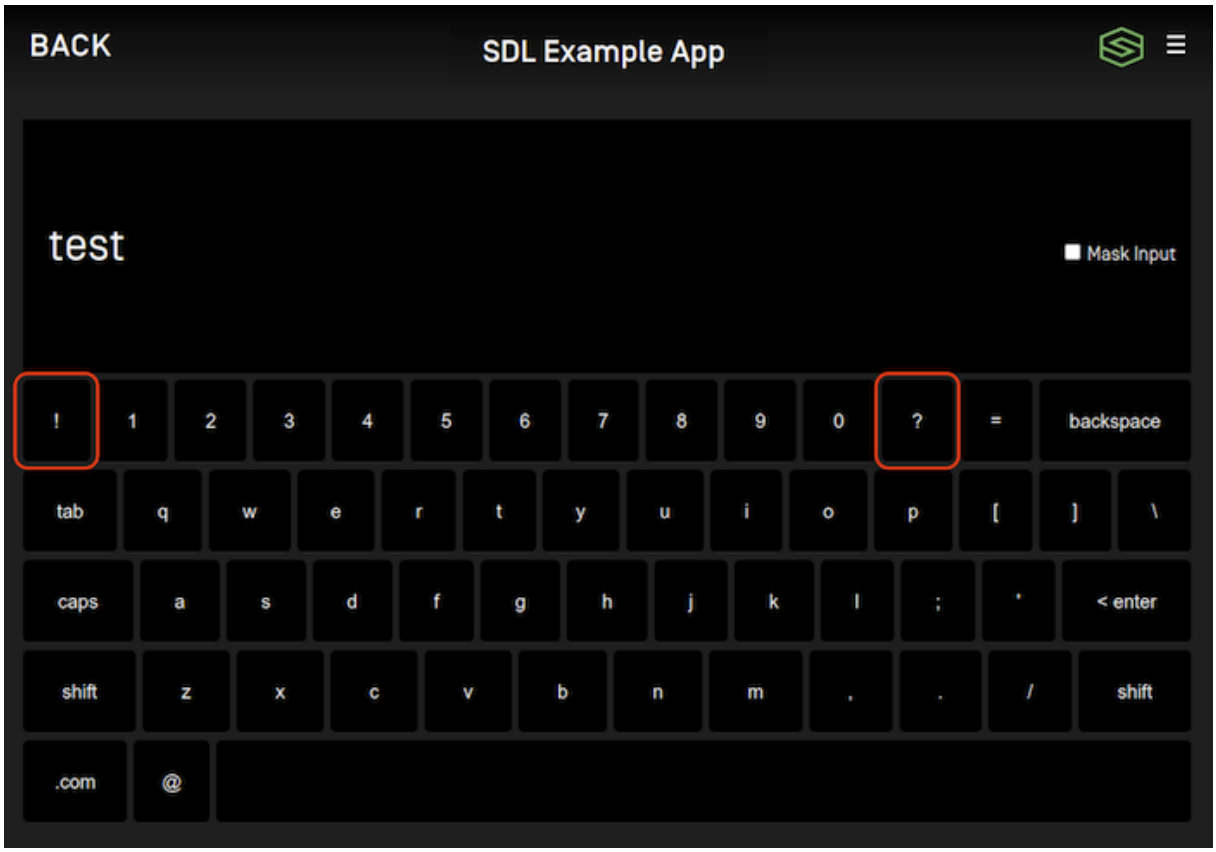
You can modify the keyboard to mask the entered characters by changing the keyboard configuration's `maskInputCharacters`.




```
const keyboardConfiguration = new SDL.rpc.structs.KeyboardProperties()
    .setKeyboardLayout(SDL.rpc.enums.KeyboardLayout.NUMERIC)

.setMaskInputCharacters(SDL.rpc.enums.KeyboardInputMask.ENABLE_INPUT_KEY_M


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

CUSTOM KEYS (RPC 7.1+)

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




```
const keyboardConfiguration = new SDL.rpc.structs.KeyboardProperties()
    .setKeyboardLayout(SDL.rpc.enums.KeyboardLayout.QWERTY)
    .setCustomKeys(['!','?']);

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

Checking Keyboard Capabilities (RPC v7.1+)

Each head unit may support different keyboard layouts and each layout can support a different number of custom keys. Head units may not support masking input. If you want

to know which keyboard features are supported on the connected head unit, you can check the `KeyboardCapabilities`:

```
const windowCapability =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability();
const keyboardCapabilities = windowCapability.getKeyboardCapabilities();

// List of layouts and number of custom keys supported by each layout
const keyboardLayouts = keyboardCapabilities.getSupportedKeyboards();

// Boolean represents whether masking is supported or not
const maskInputSupported =
keyboardCapabilities.getMaskInputCharactersSupported();
```

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.

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

Alerts and Subtle Alerts

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

Checking if the Module Supports Alerts

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

```
const isAlertAllowed =
sdlManager.getPermissionManager().isRpcAllowed(SDL.rpc.enums.FunctionID.Alert);
const isSubtleAlertAllowed =
sdlManager.getPermissionManager().isRpcAllowed(SDL.rpc.enums.FunctionID.Subtle
```

Alerts

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

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

ALERT WITH NO SOFT BUTTONS

NOTE

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

ALERT WITH SOFT BUTTONS

Creating the AlertView

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

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

TEXT

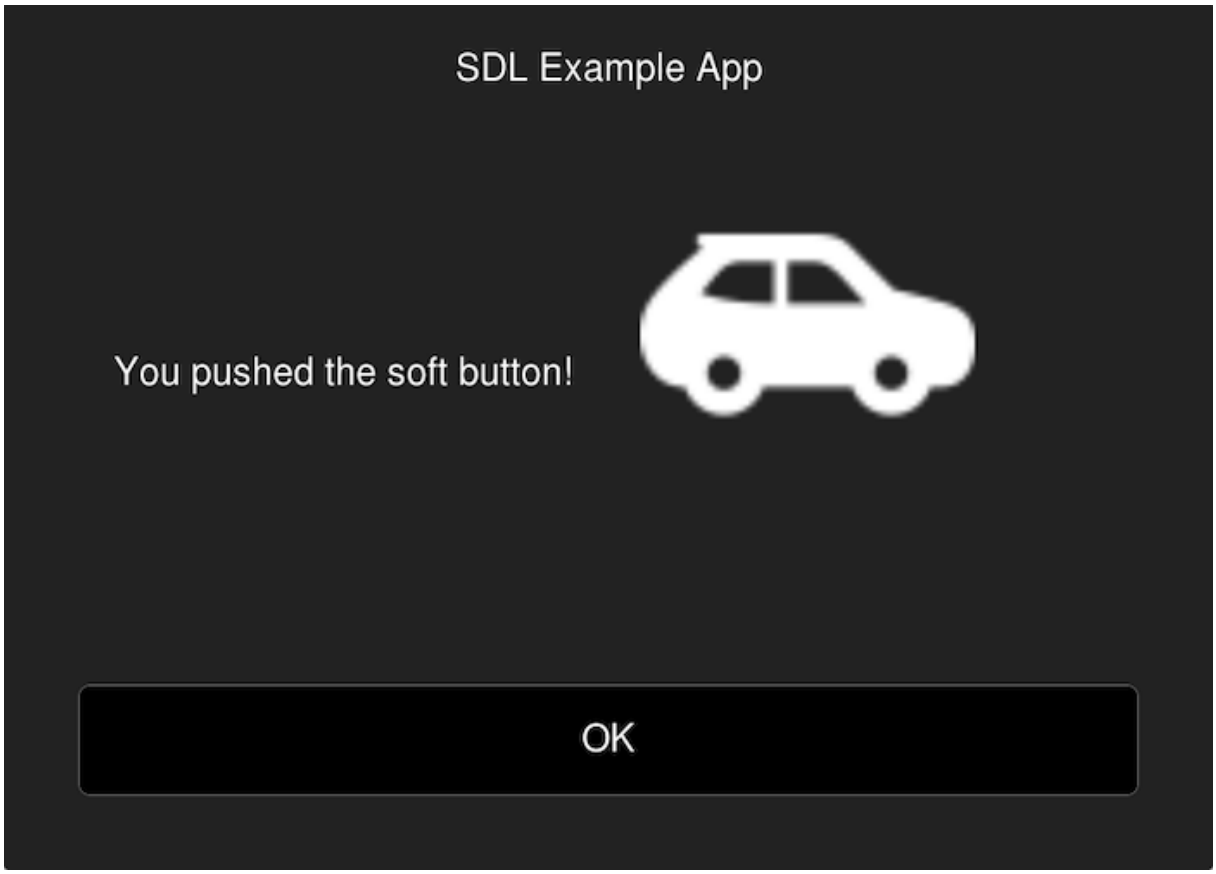
```
const alertView = new SDL.manager.screen.utils.AlertView()
    .setText('Text')
    .setSecondaryText('SecondaryText')
    .setAudio(AlertAudioData);
```

BUTTONS

```
alertView.setSoftButtons(/* List of SoftButtonObjects */);
```

ICON

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




```
alertView.setIcon(SdlArtwork);
```

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.

```
alertView.setTimeout(5);
```

PROGRESS INDICATOR

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

```
alertView.setShowWaitIndicator(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 creating an `AlertAudioData` object and setting it in the `AlertView`

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

```
const alertAudioData = new SDL.manager.screen.utils.AlertAudioData('Text to Speak')
alertView.setAudio(alertAudioData);
```

`AlertAudioData` can also play an audio file.

```
const alertAudioData = new SDL.manager.screen.utils.AlertAudioData(null, null,
sdlFile);
alertView.setAudio(alertAudioData);
```

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

```
const alertAudioData = new SDL.manager.screen.utils.AlertAudioData(null, null,
sdlFile);
const textToSpeech = [];
textToSpeech.push('Text to speak');
alertAudioData.addSpeechSynthesizerStrings(textToSpeech);
```

PLAY TONE

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

```
const alertAudioData = new SDL.manager.screen.utils.AlertAudioData('Text to
Speak')
    .setPlaytone(true);
```

Showing the Alert

```
sdlManager.getScreenManager().presentAlert(alertView, new
SDL.manager.screen.utils.AlertCompletionListener()
    .setOnComplete((success, tryAgainTime) => {
        if(success){
            // Alert was presented successfully
        }
    })
);
```

Canceling/Dismissing the Alert

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

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

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

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

```
alertView.cancel();
```

Using RPCs

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

Subtle Alerts (RPC v7.0+)

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

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

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

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

SUBTLE ALERT WITH NO SOFT BUTTONS

SUBTLE ALERT WITH SOFT BUTTONS

Creating the Subtle Alert

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

TEXT

```
const subtleAlert = new SDL.rpc.messages.SubtleAlert()
    .setAlertText1('Line 1')
    .setAlertText2('Line 2')
    .setCancelID(cancelId);
```

BUTTONS

```
// Soft buttons
const softButtonId = 10001; // Set it to any unique ID
const okButton = new SDL.rpc.structs.SoftButton()
    .setType(SDL.rpc.enums.SoftButtonType.SBT_TEXT)
    .setSoftButtonID(softButtonId)
    .setText('OK');

// Set the softbuttons(s) to the alert
subtleAlert.setSoftButtons([okButton]);

// This listener is only needed once, and will work for all of soft buttons you send
with your alert
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnButtonPress, function
(onButtonPress) {
    if (onButtonPress.getCustomButtonID() === softButtonId) {
        console.log("OK button pressed");
    }
})
```

ICON

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

```
subtleAlert.setAlertIcon(new SDL.rpc.structs.Image()
    .setValueParam('artworkName')
    .setImageType(SDL.rpc.enums.ImageType.DYNAMIC));
```

TIMEOUTS

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

```
subtleAlert.setDuration(5000);
```

TEXT-TO-SPEECH

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

```
const chunk = new SDL.rpc.structs.TTSChunk()
    .setType(SDL.rpc.enums.SpeechCapabilities.SC_TEXT)
    .setText('Text to Speak');
subtleAlert.setTtsChunks([chunk]);
```

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.

```
const ttsChunk = new SDL.rpc.structs.TTSChunk()
    .setText(sdlFile.getName())
    .setType(SDL.rpc.enums.SpeechCapabilities.FILE);
subtleAlert.setTtsChunk([ttsChunk]);
```

Showing the Subtle Alert

```
// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(subtleAlert);
if (response.getSuccess()) {
    console.log('Subtle alert was shown successfully');
}
// thrown exceptions should be caught by a parent function via .catch()
// Pre sdl_javascript_suite v1.1
// Handle RPC Response
const response = await sdlManager.sendRpc(subtleAlert).catch(function (error) {
    // Handle Error
});
if (response.getSuccess()) {
    console.log('Subtle alert was shown successfully');
}
```

Checking if the User Dismissed the Subtle Alert

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

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnSubtleAlertPressed,
function (onSubtleAlertPressed) {
    // The subtle alert was pressed
});
```

Dismissing the Subtle Alert

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

NOTE

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

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

DISMISSING A SPECIFIC SUBTLE ALERT

```
const cancelInteraction = new SDL.rpc.messages.CancelInteraction()
    .setFunctionIDParam(SDL.rpc.enums.FunctionID.SubtleAlert)
    .setCancelID(cancelID);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(cancelInteraction);
if (response.getSuccess()) {
    console.log('Subtle alert was dismissed successfully');
}
// thrown exceptions should be caught by a parent function via .catch()
// Pre sdl_javascript_suite v1.1
// Handle RPC Response
const response = await sdlManager.sendRpc(cancelInteraction).catch(function
(error) {
    // Handle Error
});
if (response.getSuccess()) {
    console.log('Subtle alert was dismissed successfully');
}
```

DISMISSING THE CURRENT SUBTLE ALERT

```
const cancelInteraction = new
SDL.rpc.messages.CancelInteraction().setFunctionIDParam(SDL.rpc.enums.FunctionI


// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(cancelInteraction);
if (response.getSuccess()) {
    console.log('Subtle alert was dismissed successfully');
}
// thrown exceptions should be caught by a parent function via .catch()
// Pre sdl_javascript_suite v1.1
// Handle RPC Response
const response = await sdlManager.sendRpc(cancelInteraction).catch(function
(error) {
    // Handle Error
});
if (response.getSuccess()) {
    console.log('Subtle alert was dismissed successfully');
}
```

Media Clock

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

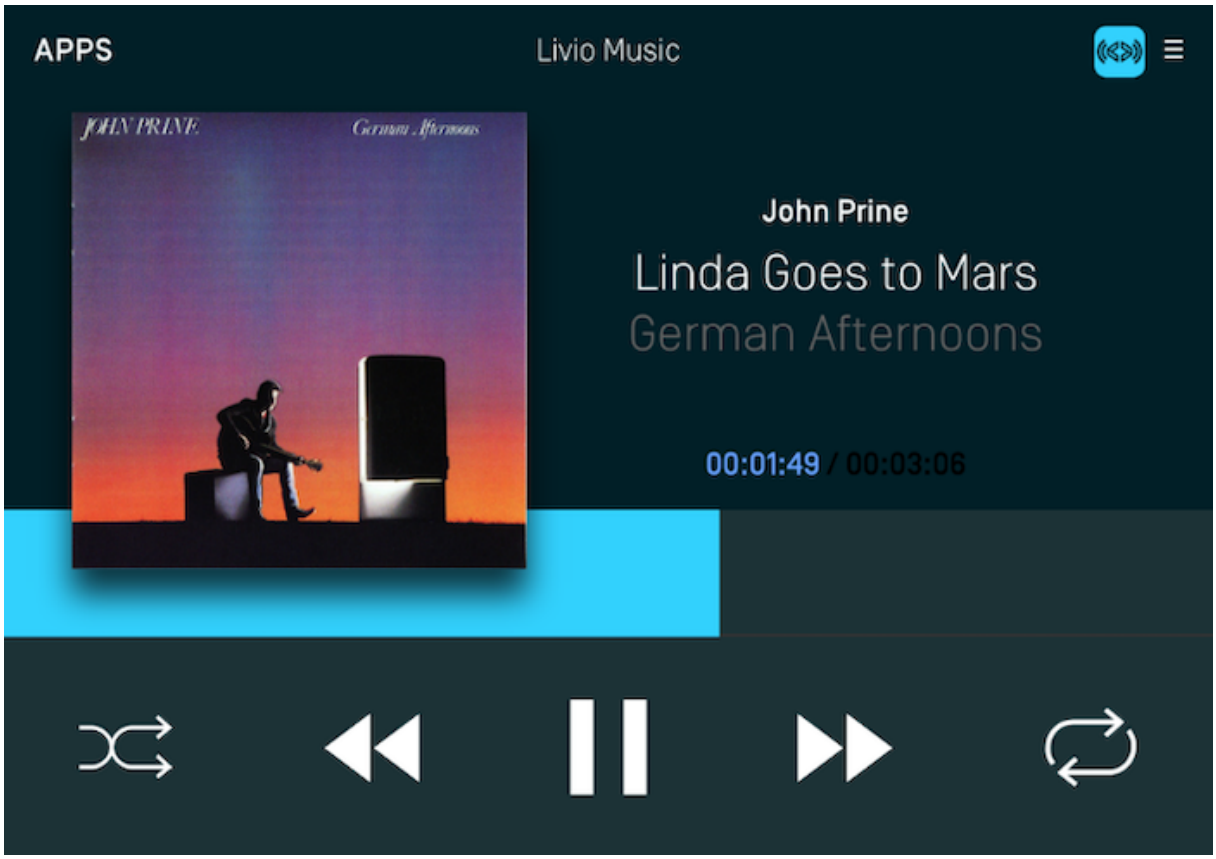
The media clock consists of three parts: the progress bar, a current position label and a remaining time label. In addition, you may want to update the play/pause button icon to reflect the current state of the audio or the media forward / back buttons to reflect if it will skip tracks or time.

NOTE

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

NOTE

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



Counting Up

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

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

```
const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.COUNTUP)
    .setStartTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(0)
            .setSeconds(30)
    ).setEndTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(4)
            .setSeconds(13)
    ).setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PAUSE);

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
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`.

```
const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.COUNTDOWN)
    .setStartTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(10)
            .setSeconds(0)
    ).setEndTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(0)
            .setSeconds(0)
    ).setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PAUSE);

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
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.

```
// Pause the progress bar and set the play / pause indicator to PLAY
const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.PAUSE)
    .setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PLAY);
// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(mediaClock);
```

```
// Resume the progress bar from its current location and set the play / pause
indicator to PAUSE
const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.RESUME)
    .setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PAUSE);
// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(mediaClock);
```

```
// Pause the progress bar, update the progress start / end time and set the play /
pause indicator to PLAY
const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.PAUSE)
    .setStartTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(1)
            .setSeconds(0)
    ).setEndTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(4)
            .setSeconds(0)
    ).setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PLAY);
// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(mediaClock);
```

Clearing the Timer

Clearing the timer removes it from the screen.

```
const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.CLEAR)
    .setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PLAY);
// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(mediaClock);
```

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

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

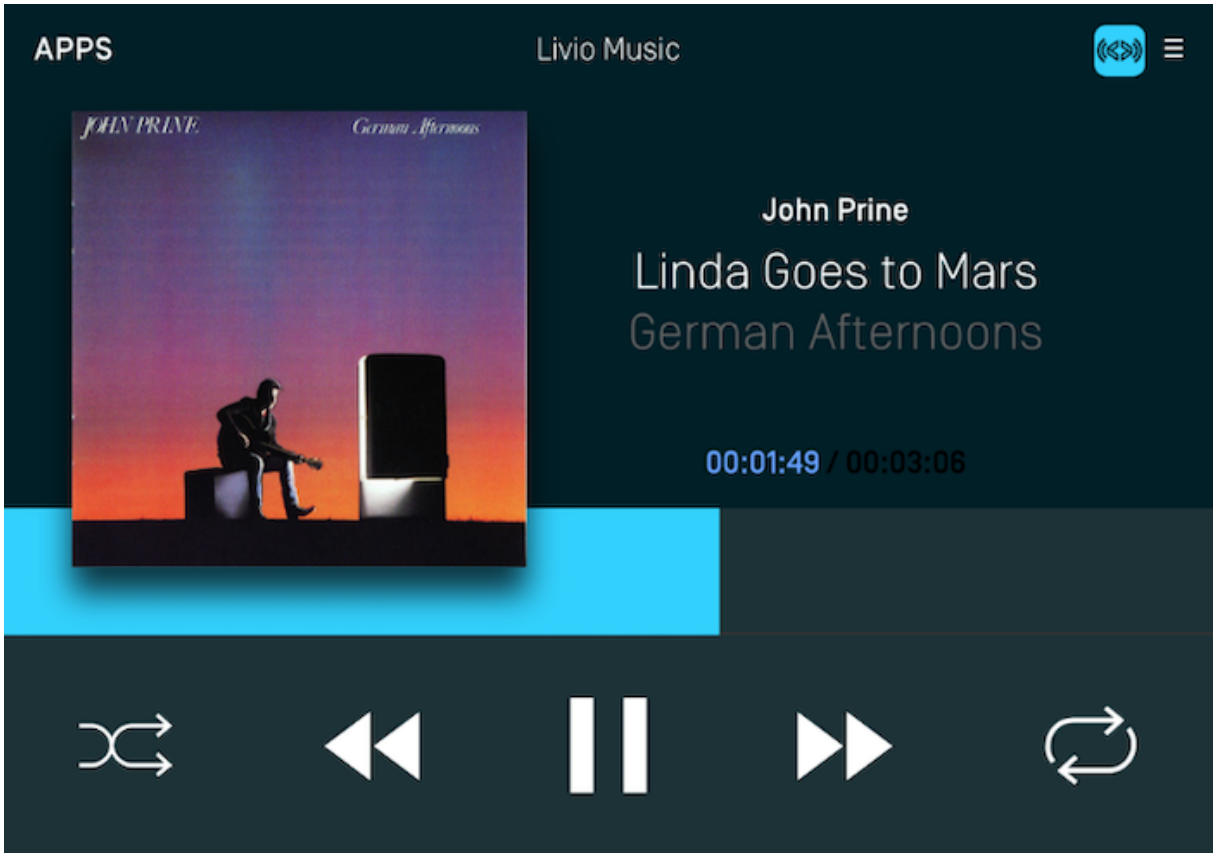
For example, a radio app will probably want two button states: play and stop. A music app, in contrast, will probably want a play and pause button. If you don't send any audio indicator information, a play / pause button will be displayed.

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

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

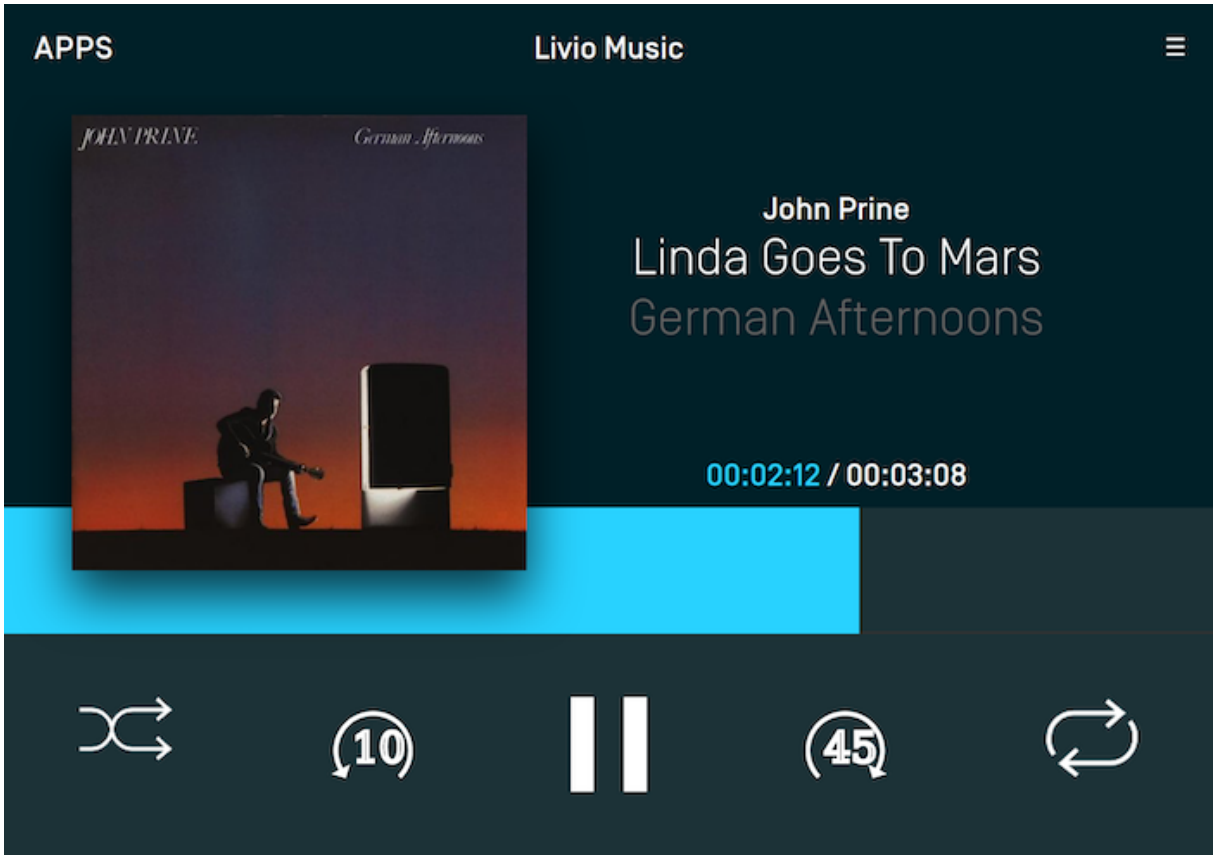
When you set the skip indicator style, you can set type `TRACK`, which is the default style that shows "skip forward" and "skip back" indicators. This is the only style available on RPC < 7.1 connections. You can also set the new type `TIME`, which will allow you to set the number of seconds and display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

Track Style




```
const streamingIndicator = new SDL.rpc.structs.SeekStreamingIndicator()
    .setType(SDL.rpc.enums.SeekIndicatorType.TRACK);

const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.PAUSE)
    .setForwardSeekIndicator(streamingIndicator)
    .setBackSeekIndicator(streamingIndicator)
    .setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PLAY);
// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(mediaClock);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(mediaClock);
```

Time Style




```
const streamingIndicator = new SDL.rpc.structs.SeekStreamingIndicator()
    .setType(SDL.rpc.enums.SeekIndicatorType.TIME)
    .setSeekTime(5);

const mediaClock = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.PAUSE)
    .setForwardSeekIndicator(streamingIndicator)
    .setBackSeekIndicator(streamingIndicator)
    .setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PLAY);
```

Adding Custom Playback Rate (RPC v7.1+)

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

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

```
//Play Audio at 50% or half speed
const mediaClockSlow = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.COUNTUP)
    .setStartTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(0)
            .setSeconds(30)
    ).setEndTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(4)
            .setSeconds(13)
    ).setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PAUSE)
    .setCountRate(0.5);

sdlManager.sendRpcResolve(mediaClockSlow);

//Play Audio at 200% or double speed
const mediaClockFast = new SDL.rpc.messages.SetMediaClockTimer()
    .setUpdateMode(SDL.rpc.enums.UpdateMode.COUNTUP)
    .setStartTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(0)
            .setSeconds(30)
    ).setEndTime(
        new SDL.rpc.structs.StartTime()
            .setHours(0)
            .setMinutes(4)
            .setSeconds(13)
    ).setAudioStreamingIndicator(SDL.rpc.enums.AudioStreamingIndicator.PAUSE)
    .setCountRate(2);

sdlManager.sendRpcResolve(mediaClockFast);
```

NOTE

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

Slider

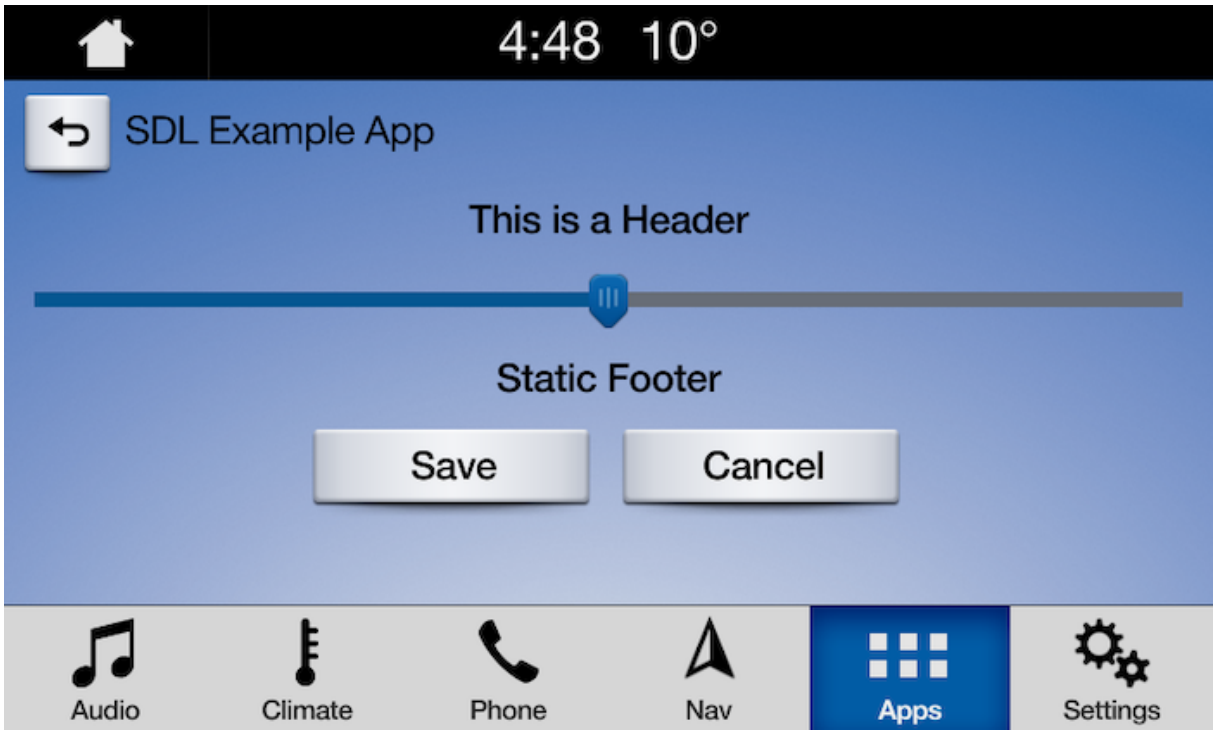
A `Slider` creates a full screen or pop-up overlay (depending on platform) that a user can control. There are two main `Slider` layouts, one with a static footer and one with a dynamic footer.

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.

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

```
const slider = new SDL.rpc.messages.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(["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 parameter. The footer message displayed will be based off of the slider's current position. The footer array should be the same length as `numTicks` because each footer must correspond to a tick value. Or, you can pass to have no footer at all.

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

Cancel ID

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

```
slider.setCancelID(10045);
```

Show the Slider

```
// sdl_javascript_suite v1.1+
const sliderResponse = await sdlManager.sendRpcResolve(slider);
if (sliderResponse.getSuccess()) {
    console.log('Slider Position Set: ' + sliderResponse.getSliderPosition());
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const sliderResponse = await sdlManager.sendRpc(slider).catch(function (error) {
    // Handle Error
});
if (sliderResponse.getSuccess()) {
    console.log('Slider Position Set: ' + sliderResponse.getSliderPosition());
}
```

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

```
// sdl_javascript_suite v1.1+
// `cancelID` is the ID that you assigned when creating the slider
const cancelInteraction = new SDL.rpc.messages.CancelInteraction()
    .setFunctionIDParam(SDL.rpc.enums.FunctionID.Slider)
    .setCancelID(cancelID);
const response = await sdlManager.sendRpcResolve(cancelInteraction);
if (response.getSuccess()) {
    console.log('Slider was dismissed successfully');
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
// `cancelID` is the ID that you assigned when creating the slider
const cancelInteraction = new SDL.rpc.messages.CancelInteraction()
    .setFunctionIDParam(SDL.rpc.enums.FunctionID.Slider)
    .setCancelID(cancelID);
const response = await sdlManager.sendRpc(cancelInteraction).catch(function
(error) {
    // Handle Error
});
if (response.getSuccess()) {
    console.log('Slider was dismissed successfully');
}
```

Dismissing the Current Slider

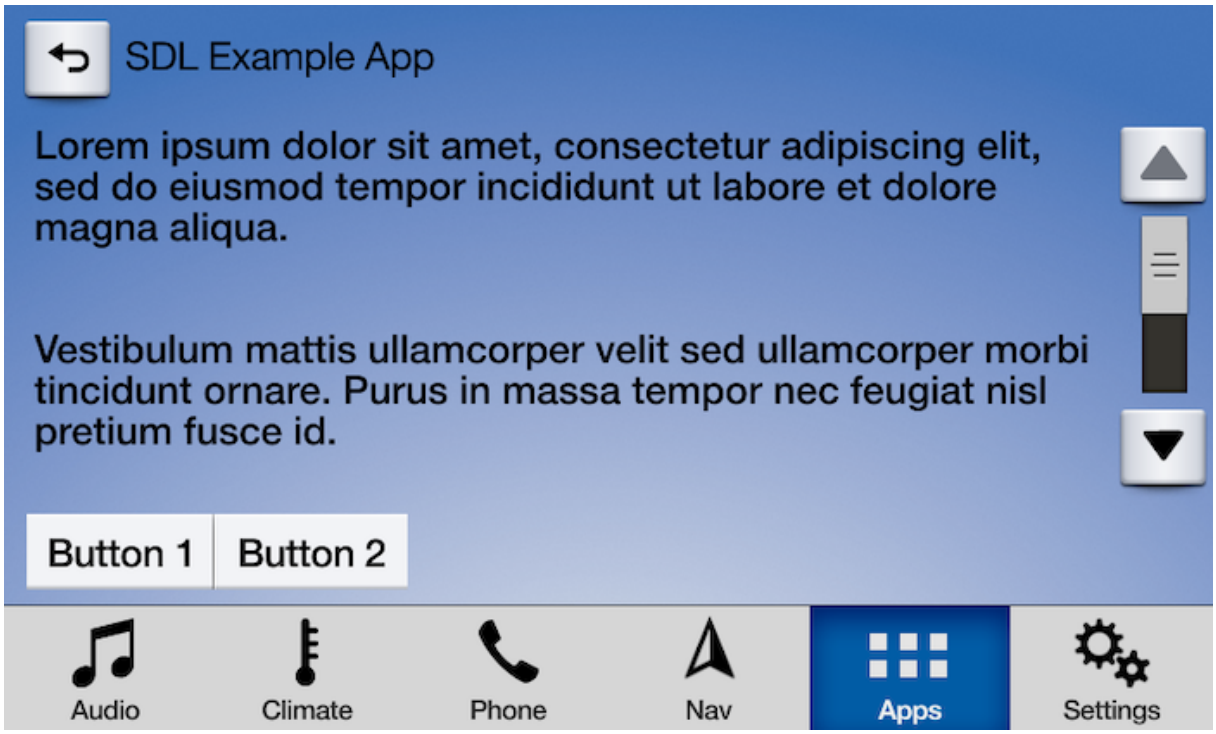
```
// sdl_javascript_suite v1.1+
const cancelInteraction = new
SDL.rpc.messages.CancelInteraction().setFunctionIDParam(SDL.rpc.enums.FunctionI

const response = await sdlManager.sendRpcResolve(cancelInteraction);
if (response.getSuccess()) {
    console.log('Slider was dismissed successfully');
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const cancelInteraction = new
SDL.rpc.messages.CancelInteraction().setFunctionIDParam(SDL.rpc.enums.FunctionI

const response = await sdlManager.sendRpc(cancelInteraction).catch(function
(error) {
    // Handle Error
});
if (response.getSuccess()) {
    console.log('Slider was dismissed successfully');
}
```

Scrollable Message

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

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

Scrollable Message UI



Creating the Scrollable Message

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

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

```
// Create Message To Display
const 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
const softButton1 = new SDL.rpc.structs.SoftButton()
    .setType(SDL.rpc.enums.SoftButtonType.SBT_TEXT)
    .setSoftButtonID(10001)
    .setText("Button 1");

const softButton2 = new SDL.rpc.structs.SoftButton()
    .setType(SDL.rpc.enums.SoftButtonType.SBT_TEXT)
    .setSoftButtonID(10002)
    .setText("Button 2");

// Create SoftButton Array
const softButtonList = [softButton1, softButton2];

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

// Set cancelId
scrollableMessage.setCancelID(integer);

// Send the scrollable message

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(scrollableMessage);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(scrollableMessage);
```

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

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnButtonPress, function
(onButtonPress) {
    switch (onButtonPress.getCustomButtonId()) {
        case 10001:
            console.log("Button 1 Pressed");
            break;
        case 10002:
            console.log("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

```
// sdl_javascript_suite v1.1+
// `cancelID` is the ID that you assigned when creating and sending the alert
const cancelInteraction = new SDL.rpc.messages.CancelInteraction()
    .setFunctionIDParam(SDL.rpc.enums.FunctionID.ScrollableMessage)
    .setCancelID(cancelID);
const response = await sdlManager.sendRpcResolve(cancelInteraction);
if (response.getSuccess()){
    console.log("Scrollable message was dismissed successfully");
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
// `cancelID` is the ID that you assigned when creating and sending the alert
const cancelInteraction = new SDL.rpc.messages.CancelInteraction()
    .setFunctionIDParam(SDL.rpc.enums.FunctionID.ScrollableMessage)
    .setCancelID(cancelID);
const response = await sdlManager.sendRpc(cancelInteraction).catch(function
(error) {
    // Handle Error
});
if (response.getSuccess()){
    console.log("Scrollable message was dismissed successfully");
}
```

Dismissing the Current Scrollable Message

```
// sdl_javascript_suite v1.1+
// `cancelID` is the ID that you assigned when creating and sending the alert
const cancelInteraction = new
SDL.rpc.messages.CancelInteraction().setFunctionIDParam(SDL.rpc.enums.FunctionI

const response = await sdlManager.sendRpcResolve(cancelInteraction);
if (response.getSuccess()){
    console.log("Scrollable message was dismissed successfully");
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const cancelInteraction = new
SDL.rpc.messages.CancelInteraction().setFunctionIDParam(SDL.rpc.enums.FunctionI

const response = await sdlManager.sendRpc(cancelInteraction).catch(function
(error) {
    // Handle Error
});
if (response.getSuccess()){
    console.log("Scrollable message was dismissed successfully");
}
```

Customizing the Template

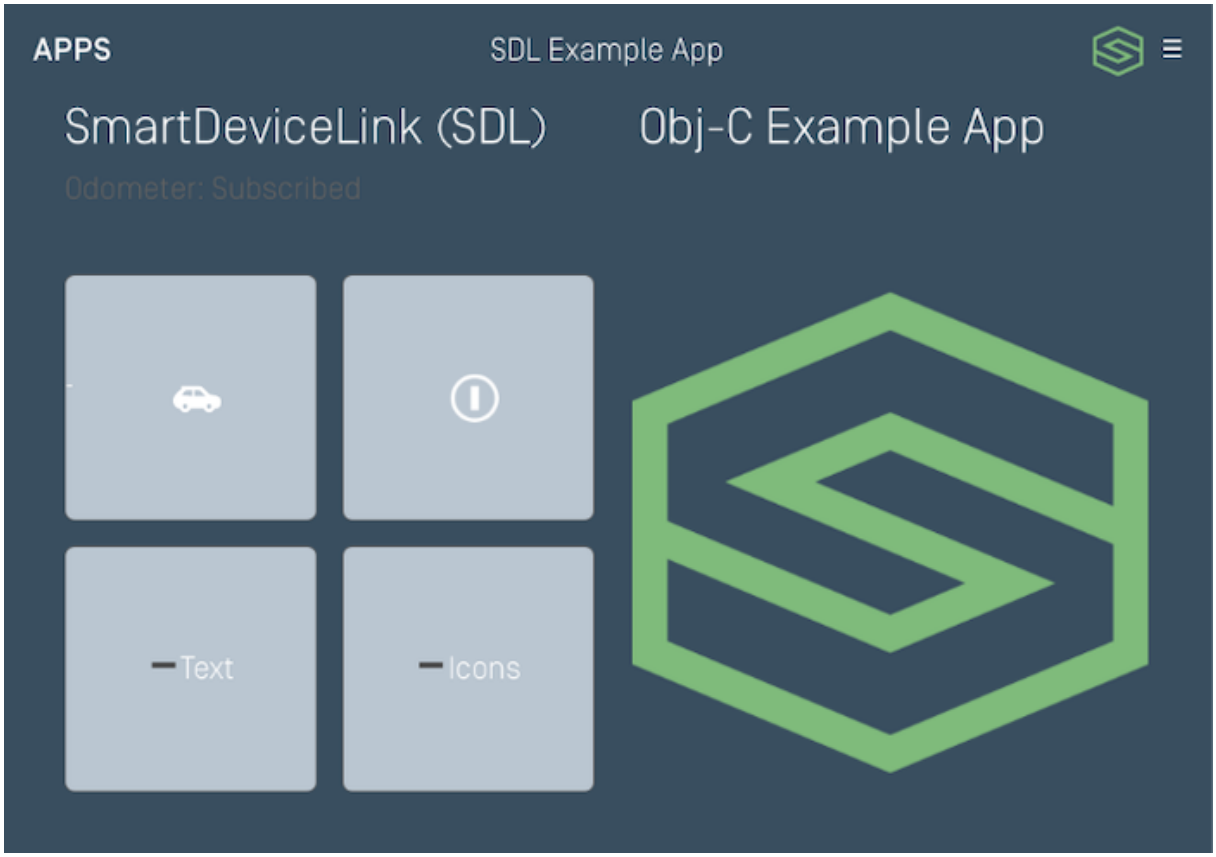
You have the ability to customize the look and feel of the template. How much customization is available depends on the RPC version of the head unit you are connected with as well as the design of the HMI.

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`.



```
// Set color schemes
const green = new
SDL.rpc.structs.RGBColor().setRed(126).setGreen(188).setBlue(121);
const white = new
SDL.rpc.structs.RGBColor().setRed(249).setGreen(251).setBlue(254);
const grey = new
SDL.rpc.structs.RGBColor().setRed(186).setGreen(198).setBlue(210);
const darkGrey = new
SDL.rpc.structs.RGBColor().setRed(57).setGreen(78).setBlue(96);

const dayColorScheme = new SDL.rpc.structs.TemplateColorScheme();
dayColorScheme.setBackgroundColor(white);
dayColorScheme.setPrimaryColor(green);
dayColorScheme.setSecondaryColor(grey);
lifecycleConfig.setDayColorScheme(dayColorScheme);

const nightColorScheme = new SDL.rpc.structs.TemplateColorScheme();
nightColorScheme.setBackgroundColor(white);
nightColorScheme.setPrimaryColor(green);
nightColorScheme.setSecondaryColor(darkGrey);
lifecycleConfig.setNightColorScheme(nightColorScheme);
```

NOTE

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

Customizing Future Layouts

You can change the template color scheme when you change layouts. This guide requires SDL JavaScript Suite version 1.2. If using an older version, use `SetDisplayLayout` (any RPC version) or `Show` (RPC v6.0+) request.

```
// Set color schemes
const green = new
SDL.rpc.structs.RGBColor().setRed(126).setGreen(188).setBlue(121);
const white = new
SDL.rpc.structs.RGBColor().setRed(249).setGreen(251).setBlue(254);
const grey = new
SDL.rpc.structs.RGBColor().setRed(186).setGreen(198).setBlue(210);
const darkGrey = new
SDL.rpc.structs.RGBColor().setRed(57).setGreen(78).setBlue(96);

const dayColorScheme = new SDL.rpc.structs.TemplateColorScheme();
dayColorScheme.setBackgroundColor(white);
dayColorScheme.setPrimaryColor(green);
dayColorScheme.setSecondaryColor(grey);
lifecycleConfig.setDayColorScheme(dayColorScheme);

const nightColorScheme = new SDL.rpc.structs.TemplateColorScheme();
nightColorScheme.setBackgroundColor(white);
nightColorScheme.setPrimaryColor(green);
nightColorScheme.setSecondaryColor(darkGrey);

const templateConfiguration = new SDL.rpc.structs.TemplateConfiguration()
    .setTemplate(SDL.rpc.enums.PredefinedLayout.GRAPHIC_WITH_TEXT)
    .setDayColorScheme(dayColorScheme)
    .setNightColorScheme(nightColorScheme);

const success = await
sdlManager.getScreenManager().changeLayout(templateConfiguration);
if (success) {
    // Color set with template change
} else {
    // Color and template not changed
}
```

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.

```
const setGlobalProperties = new SDL.rpc.messages.SetGlobalProperties();
setGlobalProperties.setMenuTitle('customTitle');
// The image must be uploaded before referencing the image name here
setGlobalProperties.setMenuIcon(image);
const response = await sdlManager.sendRpc(setGlobalProperties);
if (response.getSuccess()){
    // Success
}
```

Customizing the Keyboard (RPC v3.0+)

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

Setting Keyboard Properties

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

```
const keyboardProperties = new SDL.rpc.structs.KeyboardProperties()
    .setLanguage(SDL.rpc.enums.Language.HE_IL) // Set to Israeli Hebrew
    .setKeyboardLayout(SDL.rpc.enums.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.

```
const setGlobalProperties = new SDL.rpc.messages.SetGlobalProperties();
setGlobalProperties.setVrHelpTitle('What Can I Say?');

const item1 = new SDL.rpc.structs.VrHelpItem().setText("Show
Artists").setPosition(1).setImage(image); // a previously uploaded image or null

const item2 = new SDL.rpc.structs.VrHelpItem().setText("Show
Albums").setPosition(2).setImage(image); // a previously uploaded image or null

setGlobalProperties.setVrHelp([item1, item2]);
sdlManager.sendRpc(setGlobalProperties).catch(err => err); // If there was an error,
catch it and return it
if (response instanceof SDL.rpc.RpcRespone && response.getSuccess()) {
    // The help menu is updated
} else {
    // Handle Error
}
```

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.

```
const setGlobalProperties = new SDL.rpc.messages.SetGlobalProperties();
const chunk = new SDL.rpc.structs.TTSChunk().setText('Your custom help
prompt').setType(SDL.rpc.enums.SpeechCapabilities.SC_TEXT);
setGlobalProperties.setHelpPrompt([chunk]);
const response = await sdlManager.sendRpc(setGlobalProperties).catch(err => err);
// If there was an error, catch it and return it
if (response instanceof SDL.rpc.RpcRespone && response.getSuccess()) {
    // The help prompt is updated
} else {
    // Handle Error
}
```

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.

```
const setGlobalProperties = new SDL.rpc.messages.SetGlobalProperties();
const chunk = new SDL.rpc.structs.TTSChunk().setText('Your custom help
prompt').setType(SDL.rpc.enums.SpeechCapabilities.SC_TEXT);
setGlobalProperties.setTimeoutPrompt([chunk]);
const response = await sdlManager.sendRpc(setGlobalProperties).catch(err => err);
// If there was an error, catch it and return it
if (response instanceof SDL.rpc.RpcRespone && response.getSuccess()) {
    // The timeout prompt is updated
} else {
    // Handle Error
}
```

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
const resetGlobalProperties = new
SDL.rpc.messages.ResetGlobalProperties().setProperties([SDL.rpc.enums.GlobalProp
 SDL.rpc.enums.GlobalProperty.VRHELPTITLE]);

// Reset the menu icon and title
const resetGlobalProperties = new
SDL.rpc.messages.ResetGlobalProperties().setProperties([SDL.rpc.enums.GlobalProp
 SDL.rpc.enums.GlobalProperty.MENUNAME]);

// Reset spoken prompts
const resetGlobalProperties = new
SDL.rpc.messages.ResetGlobalProperties().setProperties([SDL.rpc.enums.GlobalProp
 SDL.rpc.enums.GlobalProperty.TIMEOUTPROMPT]);

// To send any one of these, use the typical format:
const response = await sdlManager.sendRpc(setGlobalProperties).catch(err => err);
// If there was an error, catch it and return it
if (response instanceof SDL.rpc.RpcRespone && response.getSuccess()) {
    // The global properties are reset
} else {
    // Handle Error
}
```

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 ongoing speech request. If you want to chain speech requests you must wait for the current speech request to finish before sending the next speech request.

NOTE

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

Creating the Speak Request

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

Getting the Supported Speech Capabilities

Once you have successfully connected to the module, you can access supported speech capabilities properties on the `sdlManager.getSystemCapabilityManager()` instance.

```
// This is technically a private property and a `getSpeechCapabilities` method will be added to retrieve it in a future release.
let speechCapabilities = sdlManager.getSystemCapabilityManager()._speechCapabilities;
```

Below is a list of commonly supported speech capabilities.

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

Creating Different Types of Speak Requests

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

TEXT PHRASE

```
const chunk = new
SDL.rpc.structs.TTSChunk().setText('hello').setType(SDL.rpc.enums.SpeechCapabilitie

const speak = new SDL.rpc.messages.Speak().setTtsChunks([chunk]);
```

SAPI PHONEMES PHRASE

```
const chunk = new SDL.rpc.structs.TTSChunk().setText('h eh - l ow
1').setType(SDL.rpc.enums.SpeechCapabilities.SAPI_PHONEMES);
const speak = new SDL.rpc.messages.Speak([chunk]);
```

Sending the Speak Request

```
// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(speak);
if (!response.getSuccess()){
    switch (response.getResultCode()){
        case SDL.rpc.enums.Result.DISALLOWED:
            console.log('The app does not have permission to use the speech request');
            break;
        case SDL.rpc.enums.Result.REJECTED:
            console.log('The request was rejected because a higher priority request is in
progress');
            break;
        case SDL.rpc.enums.Result.ABORTED:
            console.log('The request was aborted by another higher priority request');
            break;
        default:
            console.log('Some other error occurred');
    }
} else {
    console.log('Speech was successfully spoken');
}

// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(speak);
if (!response.getSuccess()){
    switch (response.getResultCode()){
        case SDL.rpc.enums.Result.DISALLOWED:
            console.log('The app does not have permission to use the speech request');
            break;
        case SDL.rpc.enums.Result.REJECTED:
            console.log('The request was rejected because a higher priority request is in
progress');
            break;
        case SDL.rpc.enums.Result.ABORTED:
            console.log('The request was aborted by another higher priority request');
            break;
        default:
            console.log('Some other error occurred');
    }
} else {
    console.log('Speech was successfully spoken');
}
```

Playing Audio Indications (RPC v5.0+)

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.

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

Uploading the Audio File

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

```
const audioFile = new SDL.manager.file.filetypes.SdlFile('Audio file name',
SDL.rpc.enums.FileType.AUDIO_MP3, fileData, true);
const success = await sdlManager.getFileManager().uploadFile(audioFile)
```

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.

```
const alert = new SDL.rpc.messages.Alert();
alert.setAlertText1('Alert Text 1');
alert.setAlertText2('Alert Text 2');
alert.setDuration(5000);
alert.setTtsChunks([new SDL.rpc.structs.TTSChunk().setText('Audio file
name').setType(SDL.rpc.enums.SpeechCapabilities.FILE)]);
// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(alert);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(alert);
```

Setting Up Voice Commands

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

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

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

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

Creating Voice Commands

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

```
const voiceCommand = new SDL.manager.screen.utils.VoiceCommand(['Command One'], function () {
    // Handle the VoiceCommand's Selection
});
sdlManager.getScreenManager().setVoiceCommands([voiceCommand]);
```

Unsupported Voice Commands

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

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

aborted and the previous voice commands will remain available.

Duplicate Strings in Voice Commands

DUPLICATES BETWEEN DIFFERENT COMMANDS

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

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

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

DUPLICATES IN THE SAME COMMAND

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

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

Then the manager will strip the duplicates to:

```
```

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

Deleting Voice Commands

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

```
sdlManager.getScreenManager().setVoiceCommands([]);
```

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

Using RPCs

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

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

Starting Audio Capture

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

Getting the Supported Capabilities

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

anager.getSystemCapabilityManager instance.

```
// This is technically a private property and a `getAudioPassThruCapabilities` method will be added to retrieve it in a future release.
let audioPassThruCapabilities = sdlManager.getSystemCapabilityManager()._audioPassThruCapabilities;
```

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

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

Sending the Audio Capture Request

To initiate audio capture, first construct a PerformAudioPassThru request.

```
const audioPassThru = new SDL.rpc.messages.PerformAudioPassThru()
    .setAudioPassThruDisplayText1('Ask me "What\'s the weather?"')
    .setAudioPassThruDisplayText2('or "What\'s 1 + 2?"')
    .setInitialPrompt([new SDL.rpc.structs.TTSChunk()
        .setType(SDL.rpc.enums.SpeechCapabilities.SC_TEXT)
        .setText('Ask me What\'s the weather? or What\'s 1 plus 2?')
    ])
    .setSamplingRate(SDL.rpc.enums.SamplingRate.SamplingRate_16KHZ)
    .setMaxDuration(7000)
    .setBitsPerSample(SDL.rpc.enums.BitsPerSample.BitsPerSample_16_BIT)
    .setAudioType(SDL.rpc.enums.AudioType.PCM)
    .setMuteAudio(false);

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(performAPT);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(performAPT);
```

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 app is in charge of saving previously retrieved audio data.

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnAudioPassThru, function
(onAudioPassThru) {
    if(onAudioPassThru instanceof SDL.rpc.messages.OnAudioPassThru) {
        const dataRcvd = onAudioPassThru.getBulkData();
        // Do something with current 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 four ways:

1. The audio pass thru has timed out.
    - If the audio pass thru surpasses the timeout duration, this request will be ended with a `resultCode` of `SUCCESS`. You should handle the audio pass thru as though it was successful.
2. The audio pass thru was closed due to user pressing "Cancel" (or other head-unit provided cancellation button).
    - If the audio pass thru was displayed, and the user pressed the "Cancel" button, you will receive a `resultCode` of `ABORTED`. You should ignore the audio pass thru.
3. The audio pass thru was closed due to user pressing "Done" (or other head-unit provided completion button).
    - If the audio pass thru was displayed and the user pressed the "Done" button, you will receive a `resultCode` of `SUCCESS`. You should handle the audio pass thru as though it was successful.
4. The audio pass thru was ended due to a request from the app for it to end.
    - If the audio pass thru was displayed, but you have established on your own that you no longer need to capture audio data, you can send an `EndAudioPassThru` RPC. You will receive a `resultCode` of `SUCCESS`. Depending on the reason that you sent the `EndAudioPassThru` RPC, you can choose whether or not to handle the audio pass thru as though it were successful. See Manually Stopping Audio Capture below for more details.

Manually Stopping Audio Capture

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

```
// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(performAPT);
if (response instanceof SDL.rpc.messages.PerformAudioPassThruResponse) {
    if (response.getResultCode() === SDL.rpc.enums.Result.SUCCESS) {
        // We can use the data
    } else {
        // Cancel any usage of the data
        console.log('Audio pass thru attempt failed.');
    }
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(performAPT).catch(error => error);
if (response instanceof SDL.rpc.messages.PerformAudioPassThruResponse) {
    if (response.getResultCode() === SDL.rpc.enums.Result.SUCCESS) {
        // We can use the data
    } else {
        // Cancel any usage of the data
        console.log('Audio pass thru attempt failed.');
    }
} else {
    // Handle Error
}

// The end audio pass thru was sent successfully
```

Handling the Response

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

Batch Sending RPCs

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

Both methods can have the `await` syntax be used to pause execution until all the responses return, and errors can be caught by attaching a `catch` handler. The concurrent method accepts an array of requests and will return an array of responses, while the sequential method accepts an array of requests and returns the last RPC response in the array.

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.

NOTE

The JavaScript library concurrent `sendRpcs` method will honor the ordering of the requests passed in (the method uses Promise.all behind the scenes). Each response in the array has the same position of their matching request.

```
// sdl_javascript_suite v1.1+
const subscribeButtonLeft = new SDL.rpc.messages.SubscribeButton()
    .setButtonName(SDL.rpc.enums.ButtonName.SEEKLEFT);
const subscribeButtonRight = new SDL.rpc.messages.SubscribeButton()
    .setButtonName(SDL.rpc.enums.ButtonName.SEEKRIGHT);

const responses = await sdlManager.sendRpcsResolve([subscribeButtonLeft,
subscribeButtonRight],
    (result, messagesRemaining) => {
        // this is the update callback function
    });
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const subscribeButtonLeft = new SDL.rpc.messages.SubscribeButton()
    .setButtonName(SDL.rpc.enums.ButtonName.SEEKLEFT);
const subscribeButtonRight = new SDL.rpc.messages.SubscribeButton()
    .setButtonName(SDL.rpc.enums.ButtonName.SEEKRIGHT);

const responses = await sdlManager.sendRpcs([subscribeButtonLeft,
subscribeButtonRight])
    .catch(error => {
        // if an RPC isn't successful, this is invoked with the passed-in failed RPC
    });
```

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.

```
// sdl_javascript_suite v1.1+
const choiceId = 111;
const choiceSetId = 222;
const choice = new SDL.rpc.structs.Choice()
    .setChoiceID(choiceId)
    .setMenuName('Choice title');
const createInteractionChoiceSet = new
SDL.rpc.messages.CreateInteractionChoiceSet()
    .setInteractionChoiceSetID(choiceSetId)
    .setChoiceSet([choice]);
const performInteraction = new SDL.rpc.messages.PerformInteraction()
    .setInitialText('Initial Text')
    .setInteractionMode(SDL.rpc.enums.InteractionMode.MANUAL_ONLY)
    .setInteractionChoiceSetIDList([choiceSetId]);
const response = await
sdlManager.sendSequentialRpcsResolve([createInteractionChoiceSet,
performInteraction],
    (result, messagesRemaining) => {
        // this is the update callback function
    });
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const choiceId = 111;
const choiceSetId = 222;
const choice = new SDL.rpc.structs.Choice()
    .setChoiceID(choiceId)
    .setMenuName('Choice title');
const createInteractionChoiceSet = new
SDL.rpc.messages.CreateInteractionChoiceSet()
    .setInteractionChoiceSetID(choiceSetId)
    .setChoiceSet([choice]);
const performInteraction = new SDL.rpc.messages.PerformInteraction()
    .setInitialText('Initial Text')
    .setInteractionMode(SDL.rpc.enums.InteractionMode.MANUAL_ONLY)
    .setInteractionChoiceSetIDList([choiceSetId]);
const response = await
sdlManager.sendSequentialRpcs([createInteractionChoiceSet, performInteraction])
    .catch(error => {
        // if an RPC isn't successful, this is invoked with the passed-in failed RPC
    });
```

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

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

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

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

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

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

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

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

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

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

One-Time Vehicle Data Retrieval

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

```
// sdl_javascript_suite v1.1+
const vdRequest = new SDL.rpc.messages.GetVehicleData()
    .setGearStatus(true);
const response = await sdlManager.sendRpcResolve(vdRequest);

if (response.getSuccess()) {
    const gearStatus = response.getGearStatus();
    console.log('GearStatus: ' + gearStatus);
} else {
    console.log('GetVehicleData was rejected.')
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const vdRequest = new SDL.rpc.messages.GetVehicleData()
    .getGearStatus(true);
const response = await sdlManager.sendRpc(vdRequest).catch(error => error);

if (response.getSuccess()) {
    const gearStatus = response.getGearStatus();
    console.log('GearStatus: ' + gearStatus);
} else {
    console.log('GetVehicleData was rejected.')
}
```

Subscribing to Vehicle Data

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

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

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnVehicleData,
(onVehicleDataNotification) => {
    if (onVehicleDataNotification.getGearStatus() !== null) {
        console.log('GearStatus was updated to: ' +
onVehicleDataNotification.getGearStatus());
    }
});
```

Second, send the `SubscribeVehicleData` request:

```
// sdl_javascript_suite v1.1+
const subscribeRequest = new SDL.rpc.messages.SubscribeVehicleData()
    .getGearStatus(true);
const response = await sdlManager.sendRpcResolve(subscribeRequest);
if (response.getSuccess()) {
    console.log('Successfully subscribed to vehicle data.');
} else {
    console.log('Request to subscribe to vehicle data was rejected.');
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const subscribeRequest = new SDL.rpc.messages.SubscribeVehicleData()
    .getGearStatus(true);
const response = await sdlManager.sendRpc(subscribeRequest).catch(error =>
error);
if (response.getSuccess()) {
    console.log('Successfully subscribed to vehicle data.');
} else {
    console.log('Request to subscribe to vehicle data was rejected.');
}
```

Third, the `addRpcListener` function passed in 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.

```
// sdl_javascript_suite v1.1+
const unsubscribeRequest = new SDL.rpc.messages.UnsubscribeVehicleData()
    .setGearStatus(true); // unsubscribe to GearStatus data
const response = await sdlManager.sendRpcResolve(unsubscribeRequest);
if (response.getSuccess()) {
    console.log('Successfully unsubscribed to vehicle data.');
} else {
    console.log('Request to unsubscribe to vehicle data was rejected.');
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const unsubscribeRequest = new SDL.rpc.messages.UnsubscribeVehicleData()
    .setGearStatus(true); // unsubscribe to GearStatus data
const response = await sdlManager.sendRpc(unsubscribeRequest).catch(error =>
error);
if (response.getSuccess()) {
    console.log('Successfully unsubscribed to vehicle data.');
} else {
    console.log('Request to unsubscribe to vehicle data was rejected.');
}
```

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.

```
// sdl_javascript_suite v1.1+
const vdRequest = new SDL.rpc.messages.GetVehicleData()
    .setOemCustomVehicleData('OEM-X-Vehicle-Data', true);
const response = await sdlManager.sendRpcResolve(vdRequest);
if (response.getSuccess()) {
    const CustomData = response.getOemCustomVehicleData('OEM-X-Vehicle-Data');
} else {
    console.log('GetVehicleData was rejected.')
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const vdRequest = new SDL.rpc.messages.GetVehicleData()
    .setOemCustomVehicleData('OEM-X-Vehicle-Data', true);
const response = await sdlManager.sendRpc(vdRequest).catch(error => error);
if (response.getSuccess()) {
    const CustomData = response.getOemCustomVehicleData('OEM-X-Vehicle-Data');
} else {
    console.log('GetVehicleData was rejected.')
}
```

Remote Control Vehicle Features

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

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

Why Use Remote Control?

Consider the following scenarios:

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

Supported Modules

Currently, the remote control feature supports these modules:

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

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

CLIMATE

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

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

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

RADIO

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

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

SEAT

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

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

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

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

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

AUDIO

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

LIGHT

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

HMI SETTINGS

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

Remote Control Button Presses

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

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

RC MODULE	CONTROL BUTTON
	SHUFFLE
	REPEAT

Integration

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

NOTE

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

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

```
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(SDL.rpc.e
 (remoteControlCapabilities) => {
    // Save remote control capabilities
});
```

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

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

NOTE

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

```
// Get the first climate module's information
const firstClimateModule =
remoteControlCapabilities.getClimateControlCapabilities()[0];
const climateModuleId = firstClimateModule.getModuleInfo().getModuleId();
const climateModuleLocation =
firstClimateModule.getModuleInfo().getModuleLocation();
```

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

```
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(SDL.rpc.e
 (seatLocationCapability) => {
    if (seatLocationCapability.getSeatLocations() !== null &&
seatLocationCapability.getSeatLocations().length > 0) {
        const seats = seatLocationCapability.getSeatLocations();
        // Save seat location capabilities
    }
});
```

The Grid

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

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

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

Getting Module Data

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

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

SUBSCRIBING TO MODULE DATA

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

NOTE

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

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnInteriorVehicleData,
(onInteriorVehicleData) => {
    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

```
// sdl_javascript_suite v1.1+
const getInteriorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setSubscribe(true);
const response = await sdlManager.sendRpcResolve(getInteriorVehicleData);
// This can now be used to retrieve data
// Code
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const getInteriorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setSubscribe(true);
const response = await sdlManager.sendRpc(getInteriorVehicleData).catch(error =>
error);
// This can now be used to retrieve data
// Code
```

RPC v6.0+

```
// sdl_javascript_suite v1.1+
const getInteriorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setModuleId(moduleId)
    .setSubscribe(true);
const response = await sdlManager.sendRpcResolve(getInteriorVehicleData);
// This can now be used to retrieve data
// Code
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const getInteriorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setModuleId(moduleId)
    .setSubscribe(true);
const response = await sdlManager.sendRpc(getInteriorVehicleData).catch(error =>
error);
// This can now be used to retrieve data
// 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.

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

```
// sdl_javascript_suite v1.1+
const interiorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE);
const response = await sdlManager.sendRpcResolve(interiorVehicleData);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const interiorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE);
const response = await sdlManager.sendRpc(interiorVehicleData).catch(error =>
error);
// This can now be used to retrieve data
// Code
```

RPC 6.0+

```
// sdl_javascript_suite v1.1+
const interiorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setModuleId(moduleId);
const response = await sdlManager.sendRpcResolve(interiorVehicleData);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const interiorVehicleData = new SDL.rpc.messages.GetInteriorVehicleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setModuleId(moduleId);
const response = await sdlManager.sendRpc(interiorVehicleData).catch(error =>
error);
// This can now be used to retrieve data
// Code
```

Setting Module Data

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

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

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

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

```
// sdl_javascript_suite v1.1+
const seatLocation = new SDL.rpc.messages.SetGlobalProperties()
    .setUserLocation(selectedSeat);
const response = await sdlManager.sendRpcResolve(seatLocation);
// Seat location updated#>
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const seatLocation = new SDL.rpc.messages.SetGlobalProperties()
    .setUserLocation(selectedSeat);
const response = await sdlManager.sendRpc(seatLocation).catch(error => error);
// Seat location updated#>
```

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.

```
// sdl_javascript_suite v1.1+
const getInteriorVehicleDataConsent = new
SDL.rpc.messages.GetInteriorVehicleDataConsent()
    .setModuleType(moduleType)
    .setModuleIds(moduleId);
const getInteriorVehicleDataConsentResponse = await
sdlManager.sendRpcResolve(getInteriorVehicleDataConsent);
const allowed = getInteriorVehicleDataConsentResponse.getAllowances();
// Allowed is an array of true or false values
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const getInteriorVehicleDataConsent = new
SDL.rpc.messages.GetInteriorVehicleDataConsent()
    .setModuleType(moduleType)
    .setModuleIds(moduleId);
const getInteriorVehicleDataConsentResponse = await
sdlManager.sendRpc(getInteriorVehicleDataConsent).catch(error => error);
const allowed = getInteriorVehicleDataConsentResponse.getAllowances();
// Allowed is an array of true or false values
```

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 `servic`

eArea 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

```
const temp = new SDL.rpc.struct.Temperature()
    .setUnit(SDL.rpc.enums.TemperatureUnit.FAHRENHEIT)
    .setValueParam(74.1);

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

const moduleData = new SDL.rpc.structs.ModuleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setClimateControlData(climateControlData);

const setInteriorVehicleData = new SDL.rpc.messages.SetInteriorVehicleData()
    .setModuleData(moduleData);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(setInteriorVehicleData);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(setInteriorVehicleData).catch(error =>
error);
```

RPC 6.0+

```
const temp = new SDL.rpc.struct.Temperature()
    .setUnit(SDL.rpc.enums.TemperatureUnit.FAHRENHEIT)
    .setValueParam(74.1);

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

const moduleData = new SDL.rpc.structs.ModuleData()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setModuleId(moduleId)
    .setClimateControlData(climateControlData);

const setInteriorVehicleData = new SDL.rpc.messages.SetInteriorVehicleData()
    .setModuleData(moduleData);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(setInteriorVehicleData);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(setInteriorVehicleData).catch(error =>
error);
```

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

```
const buttonPress = new SDL.rpc.messages.ButtonPress()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setButtonName(SDL.rpc.enums.ButtonName.EJECT)
    .setButtonPressMode(SDL.rpc.enums.ButtonPressMode.SHORT);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(buttonPress);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(buttonPress).catch(error => error);
```

RPC 6.0+

```
const buttonPress = new SDL.rpc.messages.ButtonPress()
    .setModuleType(SDL.rpc.enums.ModuleType.CLIMATE)
    .setButtonName(SDL.rpc.enums.ButtonName.EJECT)
    .setModuleId(moduleId)
    .setButtonPressMode(SDL.rpc.enums.ButtonPressMode.SHORT);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(buttonPress);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(buttonPress).catch(error => error);
```

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.

```
// sdl_javascript_suite v1.1+
const releaseInteriorVehicleDataModule = new
SDL.rpc.messages.ReleaseInteriorVehicleDataModule()
    .setModuleType(moduleType)
    .setModuleId(moduleId);
const response = await
sdlManager.sendRpcResolve(releaseInteriorVehicleDataModule);
// Module Was Released
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const releaseInteriorVehicleDataModule = new
SDL.rpc.messages.ReleaseInteriorVehicleDataModule()
    .setModuleType(moduleType)
    .setModuleId(moduleId);
const response = await
sdlManager.sendRpc(releaseInteriorVehicleDataModule).catch(error => error);
// Module Was Released
```

Creating an App Service (RPC v5.1+)

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

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

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

actions (such as navigating to an address with the active service or to display the temperature as provided from the active weather service).

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

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

Using an app service is covered in another guide.

App Service Types

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

Publishing an App Service

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

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.

```
const manifest = new SDL.rpc.messages.AppServiceManifest()
    .setServiceType(SDL.rpc.enums.AppServiceType.MEDIA)
    .setServiceName('My Media App') // Must be unique across app services.
    .setServiceIcon(new SDL.rpc.structs.Image()
        .setValueParam('Service Icon Name')
        .setImageType(SDL.rpc.enums.ImageType.DYNAMIC)) // Previously uploaded
service icon. This could be the same as your app icon.
    .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.
    .setRpcSpecVersion(new SDL.rpc.structs.SdlMsgVersion()
        .setMajorVersion(5)
        .setMinorVersion(0)) // An *optional* parameter that limits the RPC spec
versions you can understand to the provided version *or below*.
    .setHandledRpcs([]) // 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.
    .setMediaServiceManifest(mediaManifest); // 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.

```
const mediaManifest = new SDL.rpc.structs.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).

```
const navigationManifest = new SDL.rpc.structs.NavigationServiceManifest()
    .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`.

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

2. Publish Your Service

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

```
// sdl_javascript_suite v1.1+
const publishServiceRequest = new SDL.rpc.messages.PublishAppService()
    .setAppServiceManifest(manifest);
const response = await sdlManager.sendRpcResolve(publishServiceRequest);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const publishServiceRequest = new SDL.rpc.messages.PublishAppService()
    .setAppServiceManifest(manifest);
const response = await sdlManager.sendRpc(publishServiceRequest)
    .catch(error => error);
```

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 go to the **Getting and Subscribing to Services** section.

3. Update Your Service's Data

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

> **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

```
const mediaData = new SDL.rpc.structs.MediaServiceData();
    .setMediaTitle('Some media title')
    .setMediaArtist('Some media artist')
    .setMediaAlbum('Some album')
    .setMediaImage(new SDL.rpc.structs.Image()
        .setValueParam('Some image')
        .setImageType(SDL.rpc.enums.ImageType.DYNAMIC))
    .setPlaylistName('Some playlist')
    .setIsExplicit(true)
    .setTrackPlaybackProgress(45)
    .setTrackPlaybackDuration(90)
    .setQueuePlaybackProgress(45)
    .setQueuePlaybackDuration(150)
    .setQueueCurrentTrackNumber(2)
    .setQueueTotalTrackCount(3);

const appData = new SDL.rpc.structs.AppServiceData()
    .setServiceID(myServiceId)
    .setServiceType(SDL.rpc.enums.AppServiceType.MEDIA)
    .setMediaServiceData(mediaData);

const onAppData = new SDL.rpc.messages.OnAppServiceData()
    .setServiceData(appData);

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(onAppData);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(onAppData);
```

NAVIGATION SERVICE DATA

```
const navInstructionArt = SDL.manager.file.filetypes.SdlArtwork('turn',
SDL.rpc.enums.FileType.GRAPHIC_PNG, image, true);
// We have to send the image to the system before it's used in the app service.
const success = await sdlManager.getFileManager().uploadFile(navInstructionArt);
if (success) {
    const coordinate = new SDL.rpc.structs.Coordinate()
        .setLatitudeDegrees(42)
        .setLongitudeDegrees(43);

    const locationDetails = new SDL.rpc.structs.LocationDetails()
        .setCoordinate(coordinate);

    const navigationInstruction = new SDL.rpc.structs.NavigationInstruction()
        .setLocationDetails(locationDetails)
        .setAction(SDL.rpc.enums.NavigationAction.TURN)
        .setImage(navInstructionArt.getImageRPC());

    const dateTime = new SDL.rpc.structs.DateTime()
        .setHour(2)
        .setMinute(3)
        .setSecond(4);

    const navigationData = new SDL.rpc.structs.NavigationServiceData()
        .setTimeStamp(dateTime)
        .setInstructions([navigationInstruction]);

    const appData = new SDL.rpc.structs.AppServiceData()
        .setServiceID(myServiceId)
        .setServiceType(SDL.rpc.enums.AppServiceType.NAVIGATION)
        .setNavigationServiceData(navigationData);

    const onAppData = new SDL.rpc.messages.OnAppServiceData()
        .setServiceData(appData);

    // sdl_javascript_suite v1.1+
    sdlManager.sendRpcResolve(onAppData);
    // Pre sdl_javascript_suite v1.1
    sdlManager.sendRpc(onAppData);
}
```

WEATHER SERVICE DATA

```
const weatherImage = SDL.manager.file.filetypes.SdlArtwork('sun',
SDL.rpc.enums.FileType.GRAPHIC_PNG, image, true);
// We have to send the image to the system before it's used in the app service.
const success = await sdlManager.getFileManager().uploadFile(weatherImage);
if (success) {
    const weatherData = new SDL.rpc.structs.WeatherData()
        .setWeatherIcon(weatherImage.getImageRPC());

    const coordinate = new SDL.rpc.structs.Coordinate()
        .setLatitudeDegrees(42)
        .setLongitudeDegrees(43);

    const locationDetails = new SDL.rpc.structs.LocationDetails()
        .setCoordinate(coordinate);

    const weatherServiceData = new SDL.rpc.structs.WeatherServiceData()
        .setLocation(locationDetails);

    const appData = new SDL.rpc.structs.AppServiceData()
        .setServiceID(myServiceId)
        .setServiceType(SDL.rpc.enums.AppServiceType.WEATHER)
        .setWeatherServiceData(weatherServiceData);

    const onAppData = new SDL.rpc.messages.OnAppServiceData()
        .setServiceData(appData);

    // sdl_javascript_suite v1.1+
    sdlManager.sendRpcResolve(onAppData);
    // Pre sdl_javascript_suite v1.1
    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 `GetAppServiceDataRequest`. Then, when you get the request, you will need to respond with your app service data. Therefore, you will need to store your current service data after the most recent update using `OnAppServiceData` (see the section Update Your Service's Data).

```
// Get App Service Data Request Listener
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.GetAppServiceData,
(message) => {
    if (message.getMessageType() === SDL.rpc.enums.MessageType.request) {
        const getAppServiceData = message;

        const response = new SDL.rpc.messages.GetAppServiceDataResponse()
            .setSuccess(true)
            .setCorrelationID(getAppServiceData.getCorrelationId())
            .setResultCode(SDL.rpc.enums.Result.SUCCESS)
            .setInfo('Use to provide more information about an error')
            .setServiceData(appServiceData);

        // sdl_javascript_suite v1.1+
        sdlManager.sendRpcResolve(response);
        // Pre sdl_javascript_suite v1.1
        sdlManager.sendRpc(response);
    }
});
```

Supporting Service RPCs and Actions

5. Service RPCs

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

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

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

```
const manifest = new
SDL.rpc.structs.AppServiceManifest(SDL.rpc.enums.AppServiceType.MEDIA);
...
manifest.setHandledRpcs([SDL.rpc.enums.FunctionID.ButtonPress]);
```

```
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.ButtonPress, (message) => {
    if (message.getMessageType() === SDL.rpc.enums.MessageType.request) {
        const buttonPress = message;

        const response = new SDL.rpc.messages.ButtonPressResponse()
            .setSuccess(true)
            .setResultCode(SDL.rpc.enums.Result.SUCCESS)
            .setCorrelationID(buttonPress.getCorrelationId())
            .setInfo('Use to provide more information about an error');

        // sdl_javascript_suite v1.1+
        sdlManager.sendRpcResolve(response);
        // Pre sdl_javascript_suite v1.1
        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.addRpcListener(SDL.rpc.enums.FunctionID.PerformAppServiceInteractior
 (message) => {
    if (message.getMessageType() === SDL.rpc.enums.MessageType.request) {
        const performAppServiceInteraction = message;

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

        // A result you want to send to the consumer app.
        const response = new
SDL.rpc.messages.PerformAppServiceInteractionResponse()
            .setServiceSpecificResult('Some Result')
            .setCorrelationID(performAppServiceInteraction.getCorrelationId())
            .setInfo('Use to provide more information about an error')
            .setSuccess(true)
            .setResultCode(SDL.rpc.enums.Result.SUCCESS);

        // sdl_javascript_suite v1.1+
        sdlManager.sendRpcResolve(response);
        // Pre sdl_javascript_suite v1.1
        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+)

```
const manifest = new SDL.rpc.structs.AppServiceManifest()
    .getServiceType(SDL.rpc.enums.AppServiceType.WEATHER)
    .setWeatherServiceManifest(weatherServiceManifest);

const publishServiceRequest = new SDL.rpc.messages.PublishAppService()
    .setAppServiceManifest(manifest);

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(publishServiceRequest);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(publishServiceRequest);
```

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

```
const unpublishAppService = new SDL.rpc.messages.UnpublishAppService()
    .setServiceID(serviceId);

// sdl_javascript_suite v1.1+
sdlManager.sendRpcResolve(unpublishAppService);
// Pre sdl_javascript_suite v1.1
sdlManager.sendRpc(unpublishAppService);
```

Using App Services (RPC v5.1+)

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

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

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

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

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

Getting and Subscribing to Services

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

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`.

```
// Grab the capability once
const servicesCapabilities = await
sdlManager.getSystemCapabilityManager().updateCapability(SDL.rpc.enums.SystemC


...

// Subscribe to updates
sdlManager.getSystemCapabilityManager().addOnSystemCapabilityListener(SDL.rpc.e
 (servicesCapabilities) => {

});
```

CHECKING THE APP SERVICE CAPABILITY

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

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

if (appServices !== null) {
    appServices.forEach(anAppServiceCapability => {
        // This will tell you why a service is in the list of updates
        const 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)
        const 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.

```
// sdl_javascript_suite v1.1+
// Get service data once
const getAppServiceData = new SDL.rpc.messages.GetAppServiceData()
    .setServiceType(SDL.rpc.enums.AppServiceType.MEDIA);

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

const response = await sdlManager.sendRpcResolve(getAppServiceData);
if (response !== null && response.getSuccess()) {
    const mediaServiceData = response.getServiceData().getMediaServiceData();
}
...

// Unsubscribe from updates
const unsubscribeServiceData = new
SDL.rpc.messages.GetAppServiceData(SDL.rpc.enums.AppServiceType.MEDIA);
unsubscribeServiceData.setSubscribe(false);

sdlManager.sendRpcResolve(unsubscribeServiceData);
// thrown exceptions should be caught by a parent function via .catch()


// Pre sdl_javascript_suite v1.1
// Get service data once
const getAppServiceData = new SDL.rpc.messages.GetAppServiceData()
    .setServiceType(SDL.rpc.enums.AppServiceType.MEDIA);

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

const response = await sdlManager.sendRpc(getAppServiceData).catch(error =>
error);
if (response !== null && response.getSuccess()) {
    const mediaServiceData = response.getServiceData().getMediaServiceData();
}
...

// Unsubscribe from updates
const unsubscribeServiceData = new
SDL.rpc.messages.GetAppServiceData(SDL.rpc.enums.AppServiceType.MEDIA);
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 **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.

```
const buttonPress = new SDL.rpc.messages.ButtonPress()
    .setButtonPressMode(SDL.rpc.enums.ButtonPressMode.SHORT)
    .setButtonName(SDL.rpc.enums.ButtonName.OK)
    .setModuleType(SDL.rpc.enums.ModuleType.AUDIO);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(buttonPress);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(buttonPress).catch(error => error);
```

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).

```
const performAppServiceInteraction = new
SDL.rpc.messages.PerformAppServiceInteraction()
    .setServiceUri("sdlexample://x-callback-url/showText?x-
source=MyApp&text=My%20Custom%20String")
    .setServiceID(previousServiceId)
    .setOriginApp(appId);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(performAppServiceInteraction);
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await
sdlManager.sendRpc(performAppServiceInteraction).catch(error => error);
```

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.

```
// sdl_javascript_suite v1.1+
const weatherServiceData = appServiceData.getWeatherServiceData();

if (weatherServiceData === null || weatherServiceData.getCurrentForecast() === null ||
weatherServiceData.getCurrentForecast().getWeatherIcon() === null) {
    // The image doesn't exist, exit early
    return;
}
const currentForecastImageName =
weatherServiceData.getCurrentForecast().getWeatherIcon().getValueParam();

const getFile = new SDL.rpc.messages.GetFile()
    .setFileName(currentForecastImageName)
    .setAppServiceId(serviceId);

const getFileResponse = await sdlManager.sendRpcResolve(getFile);
const fileData = getFileResponse.getBulkData();
const sdlArtwork = new SDL.manager.file.filetypes.SdlArtwork(fileName,
FileType.GRAPHIC_PNG, fileData, false);
// Use the sdlArtwork
// thrown exceptions should be caught by a parent function via .catch()


// Pre sdl_javascript_suite v1.1
const weatherServiceData = appServiceData.getWeatherServiceData();

if (weatherServiceData === null || weatherServiceData.getCurrentForecast() === null ||
weatherServiceData.getCurrentForecast().getWeatherIcon() === null) {
    // The image doesn't exist, exit early
    return;
}
const currentForecastImageName =
weatherServiceData.getCurrentForecast().getWeatherIcon().getValueParam();

const getFile = new SDL.rpc.messages.GetFile()
    .setFileName(currentForecastImageName)
    .setAppServiceId(serviceId);

const getFileResponse = await sdlManager.sendRpc(getFile).catch(error => error);
const fileData = getFileResponse.getBulkData();
const sdlArtwork = new SDL.manager.file.filetypes.SdlArtwork(fileName,
FileType.GRAPHIC_PNG, fileData, false);
// Use the sdlArtwork
```

Calling a Phone Number

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

Checking Your App's Permissions

`DialNumber` is an RPC that is usually restricted by OEMs. As a result, a module may reject your request if your app does not have the correct permissions. Your SDL app may also be restricted to only being allowed to making a phone call when your app is open (i.e. the `hmiLevel` is non-`NONE`) or when it is the currently active app (i.e. the `hmiLevel` is `FULL`).

```
const listenerId = sdlManager.getPermissionManager().addListener([new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.DialNumber
 null)], SDL.manager.permission.enums.PermissionGroupType.ANY, function
(allowedPermissions, permissionGroupStatus) {
    if (permissionGroupStatus !==
SDL.manager.permission.enums.PermissionGroupStatus.ALLOWED) {
        // Your app does not have permission to send the `DialNumber` request for its
current HMI level
        return;
    }

    // Your app has permission to send the `DialNumber` request for its current HMI
level
});
```

Checking if the Module Supports Calling a Phone Number

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

NOTE

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

```
function isDialNumberSupported () {
    // Check if the module has phone capabilities
    if
(!sdlManager.getSystemCapabilityManager().isCapabilitySupported(SDL.rpc.enums.Sy
 {
        return false;
    }

    // Legacy modules (pre-RPC Spec v4.5) do not support system capabilities, so for
versions less than 4.5 we will assume `DialNumber` is supported if
`isCapabilitySupported()` returns true
    const sdlMsgVersion =
sdlManager.getRegisterAppInterfaceResponse().getSdlMsgVersion();
    if (sdlMsgVersion === null) {
        return true;
    }
    const rpcSpecVersion = new SDL.util.Version(sdlMsgVersion);
    if (rpcSpecVersion.isNewerThan(new SDL.util.Version(4, 5, 0)) < 0) {
        return true;
    }

    // Retrieve the phone capability
    const phoneCapability =
sdlManager.getSystemCapabilityManager().getCapability(SDL.rpc.enums.SystemCapa

    return phoneCapability !== null ? phoneCapability.getDialNumberEnabled() : false;
}
```

Sending a DialNumber Request

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

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

```
const dialNumber = new SDL.rpc.messages.DialNumber()
    .setNumber('1238675309');
const response = await sdlManager.sendRpcResolve(dialNumber);
const result = response.getResultCode();
if (result === SDL.rpc.enums.Result.SUCCESS) {
    // `DialNumber` successfully sent
} else if (result === SDL.rpc.enums.Result.REJECTED) {
    // `DialNumber` was rejected. Either the call was sent and cancelled or there is no
device connected
} else if (result === SDL.rpc.enums.Result.DISALLOWED) {
    // Your app is not allowed to use `DialNumber`
}
```

Dial Number Responses

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

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

Setting the Navigation Destination

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

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 the module from that point on using the active navigation system.

Checking Your App's Permissions

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

```
const permissionElements = [];
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.SendLocatic
 null));

const listenerId =
sdlManager.getPermissionManager().addListener(permissionElements,
SDL.manager.permission.enums.PermissionGroupType.ANY, function
(allowedPermissions, permissionGroupStatus) {
    if (permissionGroupStatus !=
SDL.manager.permission.enums.PermissionGroupStatus.ALLOWED) {
        // Your app does not have permission to send the `SendLocation` request for its
current HMI level
        return;
    }

    // Your app has permission to send the `SendLocation` request for its current HMI
level
});
```

Checking if the Module Supports Sending a Location

Since some modules will not support sending a location, you should check if the module supports this feature before trying to use it. Once you have successfully connected to the module, you can check the module's capabilities via the `sdlManager.getSystemCapability`

`Manager()` as shown in the example below. Please note that you only need to check once if the module supports sending a location, however you must wait to perform this check until you know that the SDL app has been opened (i.e. the `hmiLevel` is non-`NONE`).

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

```
async function isSendLocationSupported() {
    // Check if the module has navigation capabilities
    if
(!sdlManager.getSystemCapabilityManager().isCapabilitySupported(SDL.rpc.enums.Sy
 {
        return false;
    }

    // Legacy modules (pre-RPC Spec v4.5) do not support system capabilities, so for
versions less than 4.5 we will assume `SendLocation` is supported if
`isCapabilitySupported` returns true
    let sdlMsgVersion =
sdlManager.getRegisterAppInterfaceResponse().getSdlMsgVersion();
    if (sdlMsgVersion == null) {
        return true;
    }
    let rpcSpecVersion = new SDL.util.Version(sdlMsgVersion.getMajorVersion(),
sdlMsgVersion.getMinorVersion(), sdlMsgVersion.getPatchVersion());
    if (rpcSpecVersion.isNewerThan(new SDL.util.Version(4, 5, 0)) < 0) {
        return true;
    }

    // Retrieve the navigation capability
    let isNavigationSupported = false;
    const navCapability = await
sdlManager.getSystemCapabilityManager().updateCapability(SDL.rpc.enums.SystemC

        .catch(error => {
            throw error;
        });
    if (navCapability !== null) {
        isNavigationSupported = navCapability.getSendLocationEnabled();
    }

    return isNavigationSupported;
}
```

Using Send Location

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

```
const sendLocation = new SDL.rpc.messages.SendLocation()
    .setLatitudeDegrees(42.877737)
    .setLongitudeDegrees(-97.380967)
    .setLocationName('The Center')
    .setLocationDescription('Center of the United States');

const address = new SDL.rpc.structs.OasisAddress()
    .setSubThoroughfare('900')
    .setThoroughfare('Whiting Dr')
    .setLocality('Yankton')
    .setAdministrativeArea('SD')
    .setPostalCode('57078')
    .setCountryCode('US-SD')
    .setCountryName('United States');

sendLocation.setAddress(address);


// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(sendLocation);

// Monitor response
const result = response.getResultCode();
if (result === SDL.rpc.enums.Result.SUCCESS) {
    // SendLocation was successfully sent.
} else if (result === SDL.rpc.enums.Result.INVALID_DATA) {
    // The request you sent contains invalid data and was rejected.
} else if (result === SDL.rpc.enums.Result.DISALLOWED) {
    // Your app does not have permission to use SendLocation.
}
// thrown exceptions should be caught by a parent function via .catch()


// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(sendLocation).catch(error => error);

const result = response.getResultCode();
if (result === SDL.rpc.enums.Result.SUCCESS) {
    // `SendLocation` was successfully sent
} else if (result === SDL.rpc.enums.Result.INVALID_DATA) {
    // `SendLocation` was rejected. The request contained invalid data
} else if (result === SDL.rpc.enums.Result.DISALLOWED) {
    // Your app is not allowed to use `SendLocation`
}
```

Checking the Result of Send Location

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

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

Getting the Navigation Destination (RPC v4.1+)

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

Checking Your App's Permissions

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

```
const permissionElements = [];
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.GetWayPoin
 null));
permissionElements.push(new
SDL.manager.permission.PermissionElement(SDL.rpc.enums.FunctionID.SubscribeW
 null));

const listenerId =
sdlManager.getPermissionManager().addListener(permissionElements,
SDL.manager.permission.enums.PermissionGroupType.ANY, function
(allowedPermissions, permissionGroupStatus) {
    if
(allowedPermissions[SDL.rpc.enums.FunctionID.GetWayPoints].getIsRpcAllowed()) {
        // Your app has permission to send the `GetWayPoints` request for its current
HMI level
    } else {
        // Your app does not have permission to send the `GetWayPoints` request for its
current HMI level
    }

    if
(allowedPermissions[SDL.rpc.enums.FunctionID.SubscribeWayPoints].getIsRpcAllow
 {
        // Your app has permission to send the `SubscribeWayPoints` request for its
current HMI level
    } else {
        // Your app does not have permission to send the `SubscribeWayPoints` request
for its current HMI level
    }
});
```

Checking if the Module Supports Waypoints

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

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

```
async function isGetWaypointsSupported() {
    // Check if the module has navigation capabilities
    if
(!sdlManager.getSystemCapabilityManager().isCapabilitySupported(SDL.rpc.enums.Sy
 {
        return false;
    }

    // Legacy modules (pre-RPC Spec v4.5) do not support system capabilities, so for
versions less than 4.5 we will assume `GetWayPoints` and `SubscribeWayPoints` are
supported if `isCapabilitySupported` returns true
    let sdlMsgVersion =
sdlManager.getRegisterAppInterfaceResponse().getSdlMsgVersion();
    if (sdlMsgVersion == null) {
        return true;
    }
    let rpcSpecVersion = new SDL.util.Version(sdlMsgVersion.getMajorVersion(),
sdlMsgVersion.getMinorVersion(), sdlMsgVersion.getPatchVersion());
    if (rpcSpecVersion.isNewerThan(new SDL.util.Version(4, 5, 0)) < 0) {
        return true;
    }

    // Retrieve the navigation capability
    let isNavigationSupported = false;
    const navCapability = await
sdlManager.getSystemCapabilityManager().updateCapability(SDL.rpc.enums.SystemC

        .catch(error => {
            throw error;
        });
    if (navCapability !== null) {
        isNavigationSupported = navCapability.getGetWayPointsEnabled();
    }

    return isNavigationSupported;
}
```

Subscribing to Waypoints

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

```
// Create this method to receive the subscription callback
sdlManager.addRpcListener(SDL.rpc.enums.FunctionID.OnWayPointChange,
(onWayPointChangeNotification) => {
    // Use the waypoint data
});

// After SDL has started your connection, at whatever point you want to subscribe,
send the subscribe RPC
const subscribeWayPoints = new SDL.rpc.messages.SubscribeWayPoints();

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(subscribeWayPoints);
if (response.getSuccess()) {
    // You are now subscribed!
} else {
    // Handle the errors
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(subscribeWayPoints).catch(error =>
error);
if (response.getSuccess()) {
    // You are now subscribed
} else {
    // Handle the errors
}
```

Unsubscribing from Waypoints

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

```
// sdl_javascript_suite v1.1+
const unsubscribeWayPoints = new SDL.rpc.messages.UnsubscribeWayPoints();
const response = await sdlManager.sendRpcResolve(unsubscribeWayPoints);
if (response.getSuccess()) {
    // You are now unsubscribed!
} else {
    // Handle the errors
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const unsubscribeWayPoints = new SDL.rpc.messages.UnsubscribeWayPoints();
const response = await sdlManager.sendRpc(unsubscribeWayPoints).catch(error =>
error);
if (response.getSuccess()) {
    // You are now unsubscribed
} else {
    // Handle the errors
}
```

One-Time Waypoints Request

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

```
// sdl_javascript_suite v1.1+
const getWayPoints = new SDL.rpc.messages.GetWayPoints();
const response = await sdlManager.sendRpcResolve(getWayPoints);
if (response.getSuccess()) {
    // Use the waypoint information
} else {
    // Handle the errors
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const getWayPoints = new SDL.rpc.messages.GetWayPoints();
const response = await sdlManager.sendRpc(getWayPoints).catch(error => error);
if (response.getSuccess()) {
    // Use the waypoint data
} else {
    // Handle the errors
}
```

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 String .

```
const audioFile = new SDL.manager.file.filetypes.SdlFile('File Name',
SDL.rpc.enums.FileType.AUDIO_MP3, mp3Data, true);
const success = await sdlManager.getFileManager().uploadFile(audioFile)
    .catch(error => {
        // handle errors here
        return false;
    });
if (success) {
    // File upload successful
}
```

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.

```
const files = await sdlManager.getFileManager().uploadFiles(sdlFileList)
    .catch(err => {
        // handle errors here
    });
```

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:

```
const 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.

Overwriting Stored Files

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

```
file.setOverwrite(true);
```

Checking the Amount of File Storage Left

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

```
const 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.

```
const fileIsOnHeadUnit = sdlManager.getFileManager().getRemoteFileNames().includes(fileName);
```

Deleting Stored Files

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

```
const success = await sdlManager.getFileManager().deleteRemoteFileWithName(fileName);
```

Batch Deleting Files

```
const successes = await sdlManager.getFileManager().deleteRemoteFilesWithNames(remoteFiles);
```

Uploading Images

> **NOTE**
>
> If you use the `ScreenManager`, image uploading for template graphics, soft buttons, and menu items is handled for you behind the scenes. However, you will still need to manually upload your images if you need images in an alert, VR help lists, turn-by-turn directions, or other features not currently covered by the `ScreenManager`.
>
> The JavaScript manager does not currently handle uploading graphics for menu items. This will be included in a future version.

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

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

Checking if Graphics are Supported

Before uploading images to a head unit you should first check if the head unit supports graphics. If not, you should avoid uploading unnecessary image data. To check if graphics are supported, check the `getCapability()` method of a valid `SystemCapabilityManager` obtained from `sdlManager.getSystemCapabilityManager()` to find out the display capabilities of the head unit.

```
const imageFields =
sdlManager.getSystemCapabilityManager().getDefaultMainWindowCapability().getIma

const areImagesSupported = (imageFields.length > 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 `String`.

```
const artwork = new SDL.manager.file.filetypes.SdlArtwork('image_name',
SDL.rpc.enums.FileType.GRAPHIC_PNG, image, false);
const success = await sdlManager.getFileManager().uploadFile(audioFile)
    .catch(error => {
        // handle errors here
        return false;
    });
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 (RPC v5.1+)

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

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

User Authentication

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

Setting and Getting Cloud App Properties

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

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

NOTE

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

Setting Cloud App Properties

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

```
const cloudAppProperties = new SDL.rpc.structs.CloudAppProperties()
    .setAppID("<appId>")
    .setAuthToken("<auth token>");

const setCloudAppProperties = new SDL.rpc.messages.SetCloudAppProperties()
    .setProperties(cloudAppProperties);

// sdl_javascript_suite v1.1+
const response = await sdlManager.sendRpcResolve(setCloudAppProperties);
if (response.getSuccess()) {
    console.log("Request was successful.");
} else {
    console.log("Request was rejected.");
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const response = await sdlManager.sendRpc(setCloudAppProperties).catch(error =>
error);
if (response.getSuccess()) {
    console.log("Request was successful.");
} else {
    console.log("Request was rejected.");
}
```

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:

```
// sdl_javascript_suite v1.1+
const getCloudAppProperties = new SDL.rpc.message.GetCloudAppProperties()
    .setAppID("<appId>");

const response = await sdlManager.sendRpcResolve(getCloudAppProperties);
if (response.getSuccess()) {
    console.log("Request was successful.");
} else {
    console.log("Request was rejected.");
}
// thrown exceptions should be caught by a parent function via .catch()

// Pre sdl_javascript_suite v1.1
const getCloudAppProperties = new SDL.rpc.message.GetCloudAppProperties()
    .setAppID("<appId>");

const response = await sdlManager.sendRpc(getCloudAppProperties).catch(error =>
error);
if (response.getSuccess()) {
    console.log("Request was successful.");
} else {
    console.log("Request was rejected.");
}
```

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:

```
const 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

Encryption will be supported in a future release.

NodeJS Documentation

For Vanilla JS API Reference Documentation, please click "API Reference Documentation" below. For API Reference Documentation specific to NodeJS, see inline on GitHub.