SDL Core Guides

Document current as of 12/15/2023 02:19 PM.

Overview

Here you will find guides on how to set up SDL Core, integrate an HMI, and how to use various features in the project.

Table of Contents

Getting Started

Integrating Your HMI

Migrating to Newer SDL Versions

Developer Documentation

Feature Documentation

FAQ

Doxygen

Installation

A quick guide to installing, configuring, and running an instance of SDL Core on a Linux OS (default environment is Ubuntu 20.04 LTS).

Dependencies

The dependencies for SDL Core vary based on the configuration. You can change SDL Core's build configuration in the top level CMakeLists.txt. We have defaulted this file to a configuration which we believe is common for people who are interested in getting up and running quickly, generally on a Linux VM.

The default dependencies for SDL Core can be installed with the following command:

```
sudo apt-get install git cmake build-essential sqlite3 libsqlite3-dev libssl-dev
libssl1.1 libusb-1.0-0-dev libudev-dev libgtest-dev libbluetooth3 libbluetooth-dev bluez-
tools libpulse-dev python3-pip python3-setuptools python3-wheel python
```

Clone SDL Core and Submodules

To get the source code of SDL Core, clone the git repository like so:

```
git clone https://github.com/smartdevicelink/sdl_core
```

Before building for the first time, there are a few commands that need to be run in the source folder to initialize the project:

```
cd sdl_core
git submodule init
git submodule update
```

CMake Build Configuration

CMake is used to configure your SDL Core build before you compile the project, this is where you can enable or disable certain features such as logging. The latest list of CMake configuration options can be found in the root CMake file of the project, located at sdl_core/CMakeLists.txt. Listed below are the possible configurations for these options, default values are bolded.

TRANSPORT OPTIONS

OPTION	VALUE(S)	DEPENDENCIES	DESCRIPTION
BUILD_BT_SUPPORT	**ON**/OFF	BlueZ (packages: libbluetooth3, libbluetooth-dev, bluez-tools)	Enable/Disable bluetooth transport support via BlueZ
BUILD_USB_SUPPORT	**ON**/OFF	libusb (packages: libusb-1.0-0-dev, libudev-dev)	Enable/Disable USB transport support via libusb
BUILD_CLOUD_APP_SUPPORT	**ON**/OFF	Boost (included in project)	Enable/Disable SDL Cloud application support via boost websocket transport

FEATURE SUPPORT OPTIONS

OPTION	VALUE(S)	DEPENDENCIES	DESCRIPTION
EXTENDED_MEDIA_MODE	ON/**OFF**	GStreamer, PulseAudio (packages: libpulse-dev)	Enable/Disable audio pass thru via PulseAudio mic recording. When this option is disabled, Core will emulate audio pass thru by sending a looped audio file.
ENABLE_SECURITY	**ON**/OFF	OpenSSL (packages: libssl-dev)	Enable/Disable support for secured SDL protocol services
EXTENDED_POLICY	HTTP	N/A	HTTP (simplified) Policy flow. `OnSystemRequest` is sent with HTTP RequestType to initiate a policy table update. The HMI is not involved in the PTU process in this mode, meaning that policy table encryption is not supported.

OPTION	VALUE(S)	DEPENDENCIES	DESCRIPTION
EXTENDED_POLICY	**PROPRIETARY**	N/A	Default Policy flow, PROPRIETARY RequestType. Simplified policy feature set (no user consent, encryption/decryption only available via HMI)
EXTENDED_POLICY	EXTERNAL_PROPRIETARY	packages: python-pip, python-dev (If using the included sample policy manager, which is automatically started by `core.sh` by default)	Full Policy flow, PROPRIETARY RequestType. Full-featured policies, along with support for handling encryption/decryption via external application

DEVELOPMENT/DEBUG OPTIONS

OPTION	VALUE(S)	DEPENDENCIES	DESCRIPTION
ENABLE_LOG	**ON**/OFF	log4cxx (included in project)/boost logger	Enable/Disable logging tool. Logs are stored in `<sdl_build_dir>/bin/SmartDeviceLinkCore.log`.
LOGGER_NAME	**LOG4CXX**	log4cxx (included in project)	Build with the apache log4cxx logger. Log properties can be configured in `<sdl_build_dir>/bin/log4cxx.properties`
LOGGER_NAME	BOOST	boost logger	Build with the boost logger library. Log properties can be configured in `<sdl_build_dir>/bin/boostlogconfig.ini`
BUILD_TESTS	ON/**OFF**	GTest (packages: libgtest-dev)	Build unit tests (run with `make test`)
USE_COTIRE	**ON**/OFF	N/A	Option to use cotire to speed up the build process when BUILD_TESTS is ON.

OPTION	VALUE(S)	DEPENDENCIES	DESCRIPTION
USE_GOLD_LD	**ON**/OFF	N/A	Option to use gold linker in place of gnu ld to speed up the build process.
ENABLE_SANITIZE	ON/**OFF**	N/A	Option to compile with `-fsanitize=address` for fast memory error detection

Building

After installing the appropriate dependencies for your build configuration, you can run `cmake` with your chosen options.

Begin by creating a build folder **outside** of SDL Core source folder, for example:

```
cd ..
mkdir sdl_build
cd sdl_build
```

From the build folder you created, run `cmake {path_to_sdl_core_source_folder}` with any flags that you want to change in the format of `-D<option-name>=<value>`, for example:

```
cmake ../sdl_core
```

From there, you can build and install the project, run the following commands in your build folder:

```
make install-3rd_party
make install_python_dependencies
make install
```

For a faster build, you can run the last command with the `-j` flag, which will enable multithreaded building:

```
make -j `nproc` install
```

Start SDL Core

Once SDL Core is compiled and installed, you can start it using the provided start script in the newly created bin folder under your build folder directory

```
cd bin/
./start.sh
```

If you get a linking error when running Core, the following command may be needed to resolve it:

```
sudo ldconfig
```

In addition, you can run SDL Core as a background process using the provided daemon script. This is useful for controlling the lifecycle of Core when creating automated scripts for your system.

To start SDL Core in the background:

```
./core.sh start
```

To restart SDL Core while it is running in the background:

```
./core.sh restart
```

To stop SDL Core while it is running in the background:

```
./core.sh stop
```

To kill any lingering instances of SDL Core (including those that were not started using the script):

```
./core.sh kill
```

NOTE

If Core was built with `EXTENDED_POLICY=EXTERNAL_PROPRIETARY`, the `core.sh` script will automatically start the provided sample policy manager along with Core. To disable this, run the daemon script as such:

```
./core.sh <command> false
```

Example - EXTERNAL_PROPRIETARY build

NOTE

To perform a completely clean build after previously building SDL Core, delete the existing build folder before running these steps:

```
rm -rf sdl_build
```

The following steps can be used to build the develop branch of SDL Core from scratch with the `EXTERNAL_PROPRIETARY` policy mode enabled:

First Time Setup

The following commands only need to be run on the first installation of the project

```
sudo apt-get install git cmake build-essential sqlite3 libsqlite3-dev libssl-dev libssl1.1 libusb-1.0-0-dev libudev-dev libgtest-dev libbluetooth3 libbluetooth-dev bluez-tools libpulse-dev python3-pip python3-setuptools python3-wheel python
git clone https://github.com/smartdevicelink/sdl_core
```

Configuration

```
cd sdl_core
git checkout develop
git pull
git submodule init
git submodule update
```

Installation

```
cd ..
mkdir sdl_build
cd sdl_build
cmake ../sdl_core -DEXTENDED_POLICY=EXTERNAL_PROPRIETARY
make install-3rd_party
make install_python_dependencies
make -j3 install
```

INI Configuration

The INI file, located at `build/src/appMain/smartDeviceLink.ini` after you compile and install SDL, is where runtime options can be configured for your instance of SDL Core. Descriptions for each of these configurations are found in the file itself.

The INI file is structured as follows:

```
[section1_name]

; property1 description
property1_name = property1_value
; property2 description
property2_name = property2_value
...

[section2_name]

; property1 description
property1_name = property1_value
; property2 description
property2_name = property2_value
...

...
```

Sections

NOTE

As the guides progress, some of these sections will be discussed in greater detail.

- `HMI` - Settings relating to the HMI connection, including server and port information.
- `MEDIA MANAGER` - Settings related to media features (audio/video streaming and audio pass thru). Several of these options are described in more detail in the

Audio/Video Streaming Guide.

- `GLOBAL PROPERTIES` - Settings to define default values to set when `ResetGlobalProperties` is sent by a mobile application.
- `FILESYSTEM RESTRICTIONS` - Settings to define limits for file operations by applications in the `NONE` HMI Level.
- `AppInfo` - Settings for where to store application info for resumption purposes.
- `Security Manager` - Only used when built with ENABLE_SECURITY=ON. Settings to define how Core establishes secure services, as well as which services need to be protected.
- `Policy` - Options for policy table storage and usage.
- `TransportManager` - Configuration options for each transport adapter, including system information to be sent to SDL applications.
- `CloudAppConnections` - Only used when built with BUILD_CLOUD_APP_SUPPORT=ON. Settings for connecting to cloud applications.
- `ProtocolHandler` - SDL Protocol-level options, including the protocol version used by Core.
- `SDL5` - SDL Protocol options which were introduced with protocol version 5, allows for specifying invididivual MTUs by service type.
- `ApplicationManager` - Miscellaneous settings related to application handling.
- `Resumption` - Options regarding application resumption data storage and handling.
- `TransportRequiredForResumption` - Options for restricting HMI level resumption based on app type and transport (defined in SDL-0149).
- `LowBandwidthTransportResumptionLevel` - Extended options for restricting resumption, where exceptions can be defined for the rules in `TransportRequiredForResumption` (defined in SDL-0149).
- `MultipleTransports` - Settings related to the Multiple Transports feature, allowing an application to connect over two transports at the same time (defined in SDL-0141).
- `ServicesMap` - Settings for restricting Audio and Video services by transport, to be used in conjunction with the `MultipleTransports` section (defined in SDL-0141).
- `AppServices` - Configuration options related to the app services feature (defined in SDL-0167).
- `RCModuleConsent` - Settings regarding storage of RC module consent records.

Modifying the configuration

NOTE

SDL must be started/re-started after the `smartDeviceLink.ini` file is modified for changes to take effect.

To modify the runtime configurations for your instance of SDL Core:

1. Modify the `build/src/appMain/smartDeviceLink.ini` file
2. Re-run `make install` in the `build` directory

Setting Up Multiple Transports

The Multiple Transports feature allows apps connected to SDL Core to start another connection over a different transport for certain services. For example, an app connected over Bluetooth can use WiFi as a Secondary Transport for video streaming. This guide will walk you through how to configure the Multiple Transports feature using the `smartDeviceLink.ini` file.

Initial Setup

Modify the following lines in `smartDeviceLink.ini`.

- To enable Multiple Transports in Core:

```
[MultipleTransports]
...
MultipleTransportsEnabled = true
```

- To set the available Secondary Transport types for a given Primary Transport:

```
[MultipleTransports]
...
SecondaryTransportForBluetooth = WiFi
;SecondaryTransportForUSB =
;SecondaryTransportForWiFi =
```

NOTE

The values which can be used in the `SecondaryTransportFor` configuration are `WiFi`, `Bluetooth` and `USB`

Audio and Video Streaming

Modify the services map in `smartdeviceLink.ini` to restrict video and audio streaming services to specific transport types.

```
[ServicesMap]
...
AudioServiceTransports = TCP_WIFI
VideoServiceTransports = TCP_WIFI, AOA_USB
```

- Transports are listed in preferred order
- If a transport is not listed, then the service is not allowed to run on that transport
- If the `AudioServiceTransports`/`VideoServiceTransports` line is omitted, the corresponding service will be allowed to run on the Primary Transport

Secondary Transport Types

STRING	TYPE	DESCRIPTION
IAP_BLUETOOTH	Bluetooth	iAP over Bluetooth
IAP_USB_HOST_MODE	USB	iAP over USB, and the phone is running as host
IAP_USB_DEVICE_MODE	USB	iAP over USB, and the phone is running as device
IAP_USB	USB	iAP over USB, and Core cannot distinguish between Host Mode and Device Mode
IAP_CARPLAY	WiFi	iAP over Carplay wireless
SPP_BLUETOOTH	Bluetooth	Bluetooth SPP, either legacy SPP or SPP multiplexing
AOA_USB	USB	Android Open Accessory
TCP_WIFI	WiFi	TCP connection over Wi-Fi

Resources

For more information on how the Multiple Transports feature works, see the Feature Documentation.

SDL Core and HMI Communication

Connecting HMI to SDL

WebSocket is the primary means of communicating with the SDL Core component from the vehicle. In a basic example, an HTML5 HMI would use a native WebSocket library to communicate with SDL Core.

The HMI Adapter must:

MUST

- Be installed on the same vehicle HU OS where SDL Core is installed, or the HMI must be able to be networked to SDL Core and address it via a static IP address.
- Create and initialize components which are defined in the HMI_API specification for the version of SDL Core which is running on the vehicle HU. (For example: BasicCommunication, UI, Buttons, VR, TTS, Navigation, VehicleInfo, RC, AppService)
- Establish a separate WebSocket connection with SDL Core for each of components defined in the HMI_API specification.
- Use the appropriate corresponding connection when sending responses and notifications to any connected component.

Handshake

For opening a WebSocket connection, a handshake must be performed.

NOTE

1. Client/Server relationship
   - SDL Core is the Server
   - The HMI is the Client
2. Host
   - SDL Core is listening on **127.0.0.1:8087** by default
   - The IP and port are configurable in SDL Core's smartDeviceLink.ini file
3. WebSocket Protocol Version 13 is used by SDL Core

Example: Connecting to SDL Core with Javascript

```
connectToSDL() {
  this.socket = new WebSocket("ws://localhost:8087")
  this.socket.onopen = this.onopen.bind(this)
  this.socket.onclose = this.onclose.bind(this)
  this.socket.onmessage = this.onmessage.bind(this)
}
```

NOTE

SDL Core accepts multiple WebSocket clients and the HMI can choose to connect each interface to SDL Core via individual WebSocket connections.

Component Registration

REQUEST

The HMI must register each component which can communicate with SDL Core using the following RPC format.

KEY	VALUE INFO
id	A multiple of 100 (100, 200, 300, ...)
jsonrpc	"2.0" - constant for all messages between SDL Core and the HMI
method	"MB.registerComponent" - the request is assigned to SDL Core's MessageBroker where the component name will be associated with the socket ID. Further, SDL Core will send messages related to the named component over the corresponding connection
componentName	The name of the component being registered. Must correspond to the appropriate component name described in the current guidelines.

Example Request:

```
{
  "jsonrpc": "2.0",
  "id": 100,
  "method": "MB.registerComponent",
  "params": {
    "componentName": "BasicCommunication"
  }
}
```

The possible componentNames are:

- BasicCommunication - Generic interface containing RPCs related to HMI management. Functionality includes managing the app and device lists, opening and closing apps, SDL life cycle updates, getting system info, and system requests. This interface also contains some other one off RPCs like DialNumber and GetSystemTime.
- UI - Interface responsible for RPC events and information made visible to the user. Functionality includes getting the display capabilities, changing the app template, managing the in app menus, popups, touch events, and changing the language. It also includes the PerformAudioPassThru RPC used to capture user's speech.
- Buttons - Interface responsible for RPC events and information related to hard and soft buttons in the vehicle. Includes OnButtonPress and OnButtonEvent.
- VR - Interface responsible for RPC events and information related to voice recognition. Functionality includes managing voice commands, creating a PerformInteraction with voice commands, and notifying SDL Core when a voice recognition session begins and ends.
- TTS - Interface responsible for RPC events and information related to text to speech capabilities. Functionality includes speaking text to users, cancelling spoken text, and notifying SDL Core when a text to speech session begins and ends.
- Navigation - Interface responsible for RPC events and information related to navigation, such as audio and video streaming or interacting with the embedded navigation system by updating way points and the turn list. Includes StartStream and GetWayPoints.
- VehicleInfo - Interface responsible for RPC events and information related to vehicle data. Functionality includes retrieving the current diagnostic codes and messages, and reading vehicle type and data.
- RC - Interface responsible for RPC events and information related to the Remote Control Feature. This includes interacting with interior vehicle data such as seat, light, or radio settings within the vehicle.
- AppService - Interface responsible for RPC events and information related to the App Services Feature. This includes publishing and activating an app service, getting app service data, performing an app service interaction, and getting app service consent or records.

RESPONSE

SDL provides a JSON Response

KEY	VALUE INFO
id	The value from the corresponding request
result	Value of id multiplied by 10. HMI can treat this as a successful registration

Example Response:

```
{
  "id": 100,
  "jsonrpc": "2.0",
  "result": 1000
}
```

Component Readiness Requests

Once the components are registered, the HMI must notify SDL Core that it is ready to begin further communication using the BasicCommunication.OnReady notification.

Upon receipt of the OnReady notification, SDL Core will begin checking the availability of the different HMI components via a chain of requests:

- `UI.IsReady` - The display availability
- `VR.IsReady` - The voice recognition module availability
- `TTS.IsReady` - The text to speech module availability
- `Navigation.IsReady` - Navigation engine availability
- `VehicleInfo.IsReady` - Indicates whether vehicle information can be collected and provided
- `RC.IsReady` - Indicates whether vehicle RC modules are present and ready to communicate with SDL Core

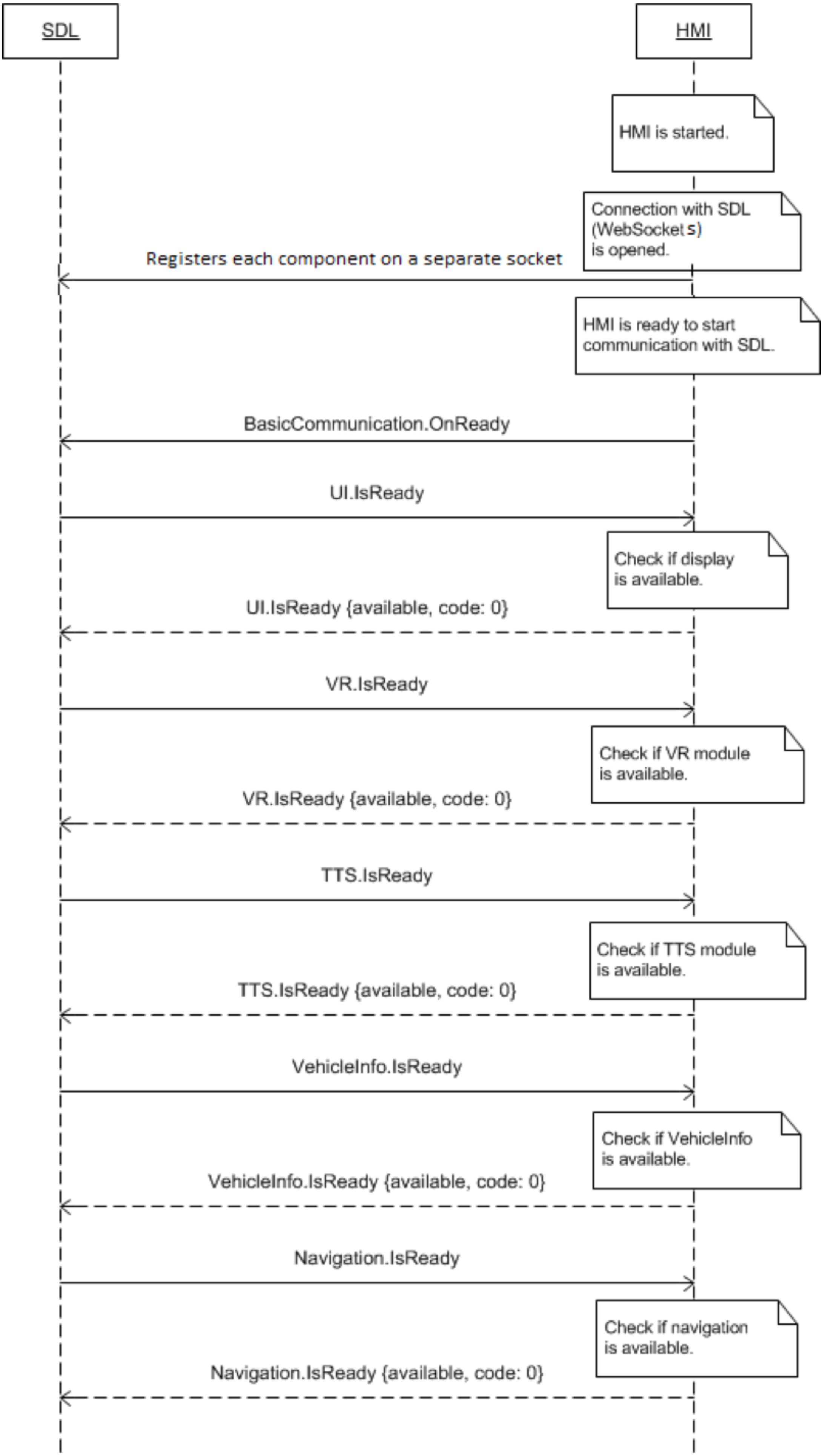
NOTE

In the case of a WebSocket connection, RPCs to each of the components are sent within a separate WebSocket connection.

SEQUENCE DIAGRAM

IsReady Sequence

View Diagram


SDL
HMI
HMI is started.
Connection with SDL (WebSocket s) is opened.
Registers each component on a separate socket
HMI is ready to start communication with SDL.
BasicCommunication.OnReady
UI.IsReady
Check if display is available.
UI.IsReady {available, code: 0}
VR.IsReady
Check if VR module is available.
VR.IsReady {available, code: 0}
TTS.IsReady
Check if TTS module is available.
TTS.IsReady {available, code: 0}
VehicleInfo.IsReady
Check if VehicleInfo is available.
VehicleInfo.IsReady {available, code: 0}
Navigation.IsReady
Check if navigation is available.
Navigation.IsReady {available, code: 0}

NOTE

If the response to any of the component `IsReady` requests contains `{"available": false}`, SDL Core will no longer communicate with that component.

Respond to BasicCommunication.GetSystemInfo

Communicating the current version of the HMI integration (CCPU) is needed for SDL Core to know when to request an update to the HMI's capabilities that may have changed since the previous software version. Core will not mark the HMI as cooperating until this response is sent by the HMI.

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": rpc.id,
  "result": {
    "method": "BasicCommunication.GetSystemInfo",
    "code": 0,
    "ccpu_version": "0.0.1",
    "language": "EN-US",
    "wersCountryCode": "WAEGB",
  }
}
```

Registering for Notifications

The HMI must also register for notifications individually using the following RPC format.

```
{
  "jsonrpc": "2.0",
  "id": -1,
  "method": "MB.subscribeTo",
  "params": {
    "propertyName": <NotificationName>
  }
}
```

"propertyName" is the name of the notification the HMI will receive from Core. Some examples include:

- Buttons.OnButtonSubscription
- BasicCommunication.OnAppRegistered
- BasicCommunication.OnAppUnregistered
- Navigation.OnVideoDataStreaming
- SDL.OnStatusUpdate

Core's MessageBroker will not route notifications to the HMI unless the notifications are subscribed to.

MUST

The HMI must:

- Register its components
- Send the OnReady notification
- Respond to each of the `IsReady` RPCs
- Register for the notifications it would like to receive

The above steps should only occur once per life cycle of SDL Core

Communicating with SDL Core

This section describes the message structure for communication between your HMI and SDL Core.

From this point forward the actors for exchanging messages will be considered:
- **Client** - can send requests and notifications
- **Server** - can provide responses to requests from a Client and send notifications

Request

An RPC call is represented by sending a Request object to a Server. The Request object has the following properties

PROPERTY	DESCRIPTION
id	An identifier established by the Client. This value must be of unsigned int type in the frames of communication between your HMI and SDL Core. The value should never be null. If "id" is not included the message is assumed to be a notification and the receiver should not respond.
jsonrpc	A string specifying the version of JSON RPC protocol being used. Must be exactly **"2.0"** currently in all versions of SDL Core.
method	A String containing the information of the method to be invoked. The format is `[componentName].[methodName]`.
params	A structured object that holds the parameter values to be used during the invocation of the method. This property may be omitted.

Example Requests

REQUEST WITH NO PARAMETERS

```
{
  "id": 125,
  "jsonrpc": "2.0",
  "method": "Buttons.GetCapabilities"
}
```

REQUEST WITH PARAMETERS

```
{
  "id": 92,
  "jsonrpc": "2.0",
  "method": "UI.Alert",
  "params": {
    "alertStrings": [
      {
        "fieldName": "alertText1",
        "fieldText": "WARNING"
      },
      {
        "fieldName": "alertText2",
        "fieldText": "Adverse Weather Conditions Ahead"
      }
    ],
    "duration": 4000,
    "softButtons": [
      {
        "type": "TEXT",
        "text": "OK",
        "softButtonID": 697,
        "systemAction": "STEAL_FOCUS"
      }
    ],
    "appID": 8218
  }
}
```

Notification

A notification is a Request object without an `id` property. For all the other properties, see the Request section above.

The receiver should not reply to a notification, i.e. no response object needs to be returned to the client upon receipt of a notification.

Example Notifications

NOTIFICATION WITH NO PARAMETERS

```
{
  "jsonrpc": "2.0",
  "method": "UI.OnReady"
}
```

NOTIFICATIONS WITH PARAMETERS

```
{
  "jsonrpc": "2.0",
  "method": "BasicCommunication.OnAppActivated",
  "params": {
    "appID": 6578
  }
}

{
  "jsonrpc": "2.0",
  "method": "Buttons.OnButtonPress",
  "params": {
    "mode": "SHORT",
    "name": "OK"
  }
}
```

Response

On receipt of a request message, the server must reply with a Response. The Response is expressed as a single JSON Object with the following properties.

MUST

An RPC must be sent in result format for its parameters to be passed to mobile.

PROPERTY	DESCRIPTION
id	Required property which must be the same as the value of the associated request object. If there was an error in detecting the id in the request object, this value must be null.
jsonrpc	Must be exactly **"2.0"**
result	The result property must contain a `method` field which is the same as the corresponding request and a corresponding result code should be sent in the result property. The result property may also include additional properties as defined in the HMI API.

Example Responses

RESPONSE WITH NO PARAMETERS

```
{
  "id": 167,
  "jsonrpc": "2.0",
  "result": {
    "code": 0,
    "method": "UI.Alert"
  }
}
```

RESPONSE WITH PARAMETERS

```
{
  "id": 125,
  "jsonrpc": "2.0",
  "result": {
    "capabilities" : [
      {
        "longPressAvailable" : true,
        "name" : "PRESET_0",
        "shortPressAvailable" : true,
        "upDownAvailable" : true
      },
      {
        "longPressAvailable" : true,
        "name" : "TUNEDOWN",
        "shortPressAvailable" : true,
        "upDownAvailable" : true
      }
    ],
    "presetBankCapabilities": {
      "onScreenPresetsAvailable" : true
    },
    "code" : 0,
    "method" : "Buttons.GetCapabilities"
  }
}
```

Error Response

MUST

An RPC must be sent in error format for its message to be passed to mobile.

The error object has the following members:

PROPERTY	DESCRIPTION
id	Required to be the same as the value of "id" in the corresponding Request object. If there was an error in detecting the id of the request object, then this property must be null.
jsonrpc	Must be exactly "2.0"
error	The error field must contain a `code` field with the result code value that indicates the error type that occurred, a `data` field with the `method` from the original request, and optionally a `message` field containing the string that provides a short description of the error.

Examples

RESPONSE WITH ERROR

```
{
  "id": 103,
  "jsonrpc": "2.0",
  "error": {
    "code": 13,
    "message": "One of the provided IDs is not valid",
    "data": {
      "method": "VehicleInfo.GetDTCs"
    }
  }
}
```

RESPONSE WITH WARNINGS AND MESSAGE

```
{
  "id": 103,
  "jsonrpc": "2.0",
  "error": {
    "code": 21,
    "message": "Requested image was not found.",
    "data": {
      "method": "UI.Alert"
    }
  }
}
```

Required Get Capability Responses

As of SDL Core 7.0, SDL Core has the ability to cache certain HMI capabilities and restore them each ignition cycle. On the first time SDL Core is started, or when the HMI's CCPU version changes, SDL Core will request the following messages to the HMI:

- UI.GetLanguage
- UI.GetSupportedLanguage
- UI.GetCapabilities
- RC.GetCapabilities
- VR.GetLanguage
- VR.GetSupportedLanguages
- VR.GetCapabilities
- TTS.GetLanguage
- TTS.GetSupportedLanguages
- TTS.GetCapabilities
- Buttons.GetCapabilities
- VehicleInfo.GetVehicleType

NOTE

If your HMI implementation registers a component (UI, RC, VR, etc), the HMI must respond to the applicable capability requests from Core.

Greater detail about each of these HMI RPCs can be found in the HMI API Reference Documentation.

Creating the HMI UI Component

Before starting the development of the SDL HMI user interface, there are a few RPC prerequisites that are required.

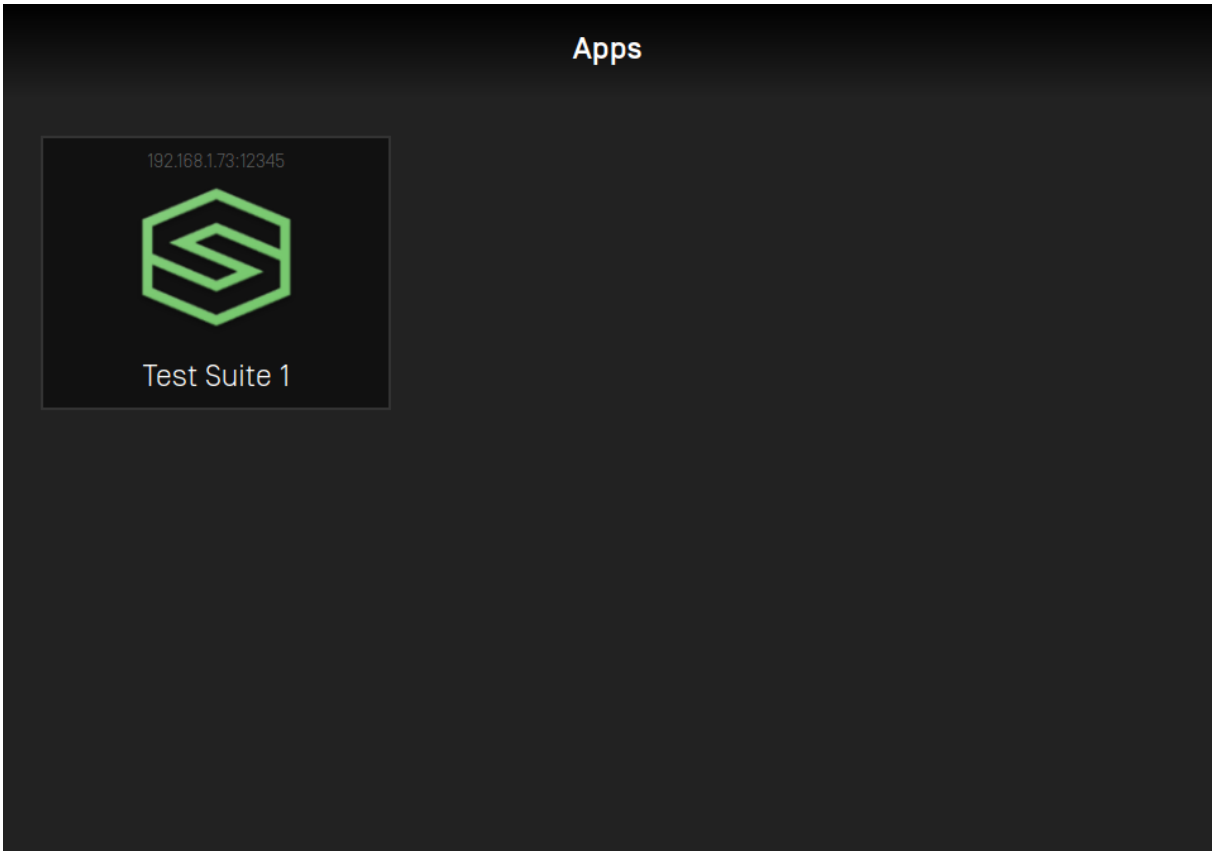
The minimum prerequisites to connect your SDL compatible user interface are:

1. Establish an HMI websocket connection to SDL Core.
2. Register the following components: BasicCommunication, Buttons, and UI.
3. Send the `BasicCommunication.OnReady` notification to SDL Core.
4. Respond to the `IsReady` request for each registered component.
5. Subscribe to the following Core notifications:
    - `BasicCommunication.OnAppRegistered`
    - `BasicCommunication.OnAppUnregistered`
    - `BasicCommunication.OnPutFile`
    - `Buttons.OnButtonSubscription`

Creating the App List

When there are changes to the list of registered apps, Core will send a `BasicCommunication.UpdateAppList` RPC request to the HMI. This request contains an array of information for all connected and pending applications. The HMI should use the information provided in this request to update its internal app list state and app list display.

For each app listed in the `UpdateAppList` request, the HMI's app list view should show a button that includes the app's name and icon.

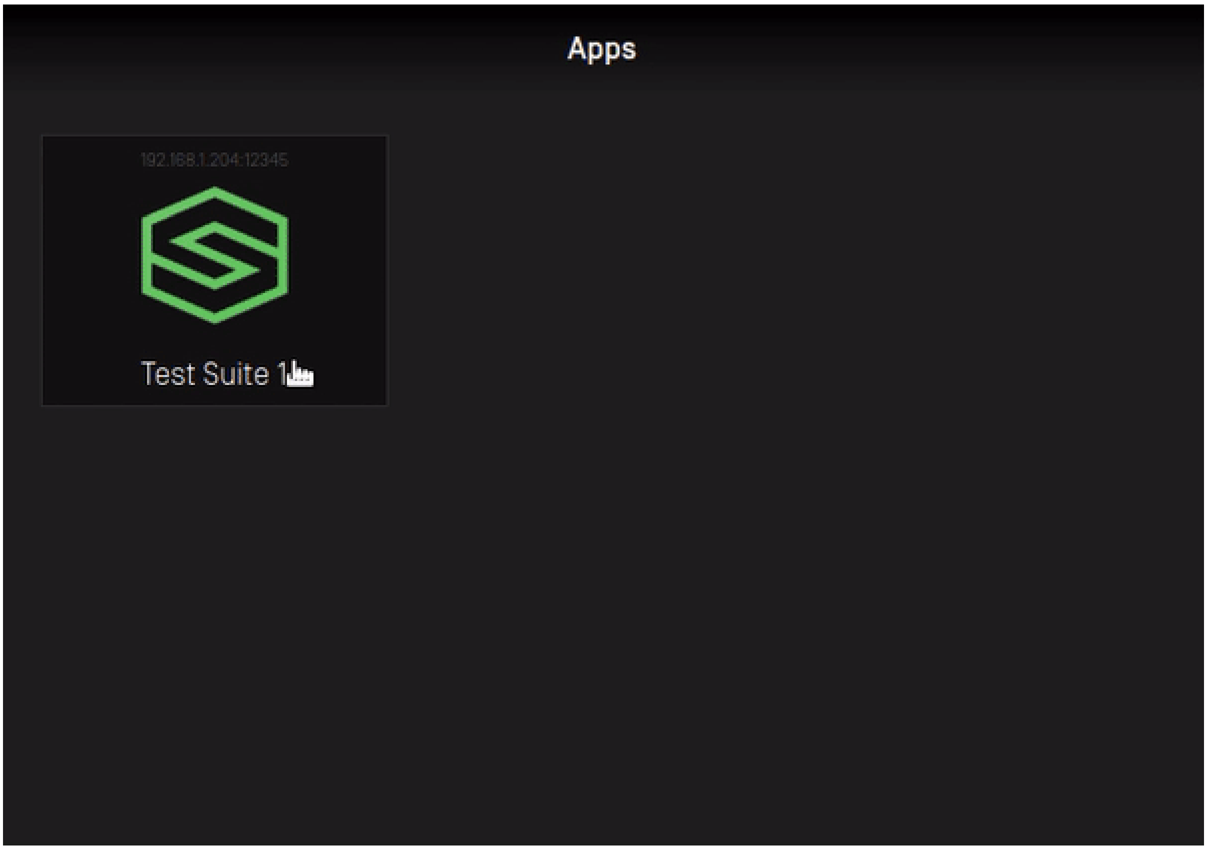


If an app is disconnected or unregistered, Core will send an `UpdateAppList` request to the HMI with that application omitted from the app list array. The HMI should make sure its app list is always up to date, and only show applications that were included in the most recent `UpdateAppList` request.

Activating an Application

User Selection

When the user selects an application from the app list, a request should be made to bring this app to the foreground (this is called "activating" the application). The first step required by the HMI when an application is selected is to send a `SDL.ActivateApp` request to Core. When Core responds with a successful `SDL.ActivateApp` response, the HMI can switch views from the app list to the app's default template.



NOTE

The default template for an app should be used if the app has not requested to use a specific template via the `UI.Show.templateConfiguration` parameter.

The default template for media apps is `MEDIA`, and the default template for all other apps is `NON-MEDIA`.

You can check if a given app is a media application using that app's `isMediaApplication` parameter, sent in the `BasicCommunication.UpdateAppList` request.

User Consent

If the `SDL.ActivateApp` response returns with the parameter `isPermissionsConsentNeeded = true`, the HMI should send a `SDL.GetListOfPermissions` request. This happens when the activating app requires permissions that the user must provide consent for. For example, if an app wants to access vehicle data, an SDL policy configuration might require the user to provide consent before the app can collect this information.

After receiving the list of permissions for the app, the HMI should show the user the `PermissionItem` name and status for each requested permission. If available, the HMI should also show a consent prompt that contains a user friendly message describing what the user is agreeing to. The user should have the ability to enable or disable each permission item. If any permission changes are made by the user, these updates should be communicated to Core via the `SDL.OnAppPermissionConsent` notification.

NOTE

Permissions are managed by SDL Core's policy table. Refer to the SDL Overview Policy Guide.

OEM defined consent prompts can be retrieved from the policy table via a `BasicCommunication.GetUserFriendlyMessage` RPC.

Resumption

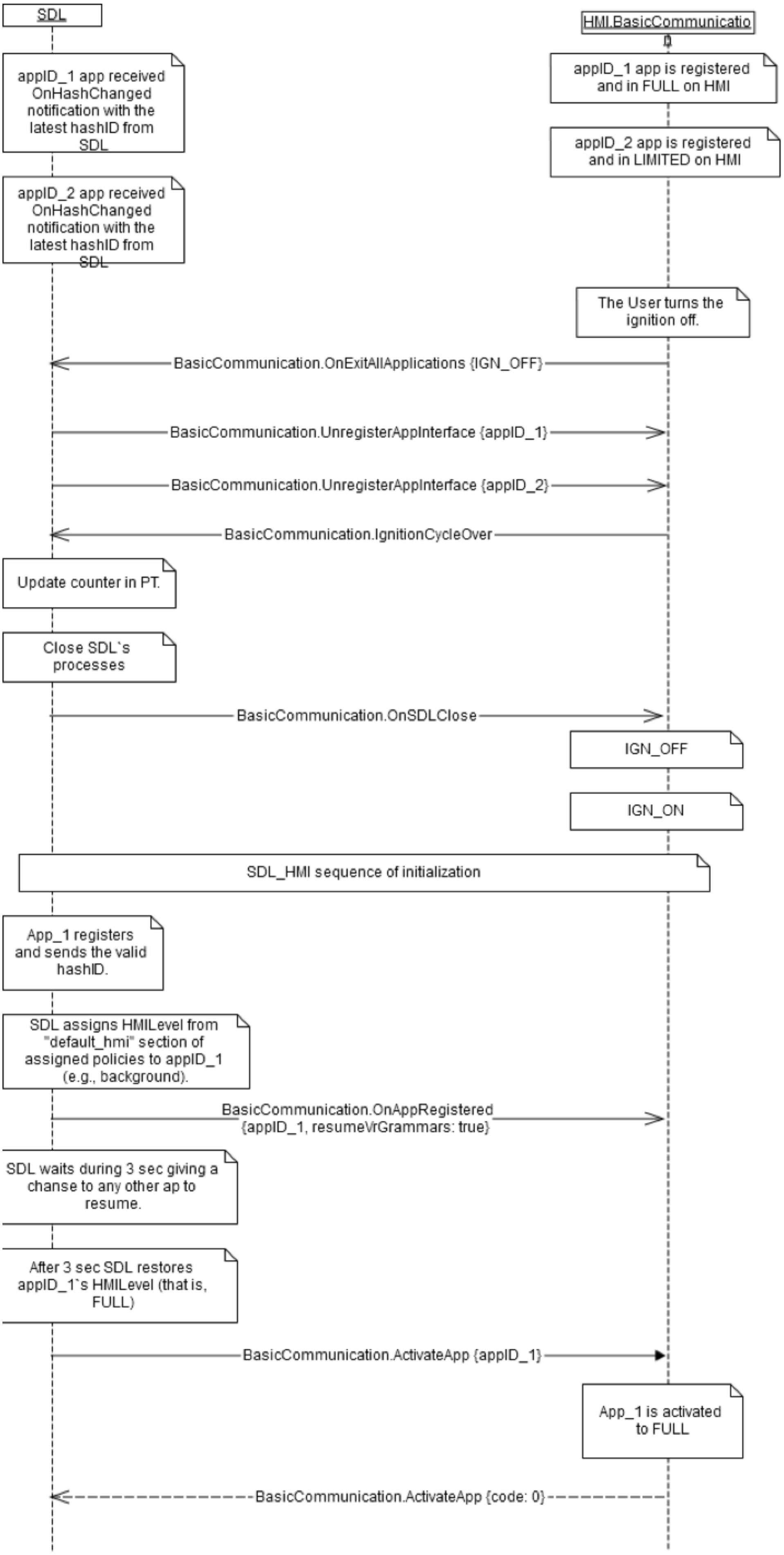
If an app is disconnected from SDL Core and reconnects within a specified time limit, Core will try to resume the app into the same HMI state the app was in before it was disconnected. The HMI should be prepared to handle a `BasicCommunication.ActivateApp` request from SDL Core, in which case the HMI should return the app into the requested state (or respond with an error if unable to). For example, if the requested HMI level is `FULL`, the HMI should activate the app and put that app's template into view.

Refer to the following resumption sequence diagram

SEQUENCE DIAGRAM

Resumption after ignition cycle

View Diagram


SDL
HMI.BasicCommunicatio
n
appID_1 app received OnHashChanged notification with the latest hashID from SDL
appID_1 app is registered and in FULL on HMI
appID_2 app is registered and in LIMITED on HMI
appID_2 app received OnHashChanged notification with the latest hashID from SDL
The User turns the ignition off.
BasicCommunication.OnExitAllApplications {IGN_OFF}
BasicCommunication.UnregisterAppInterface {appID_1}
BasicCommunication.UnregisterAppInterface {appID_2}
BasicCommunication.IgnitionCycleOver
Update counter in PT.
Close SDL`s processes
BasicCommunication.OnSDLClose
IGN_OFF
IGN_ON
SDL_HMI sequence of initialization
App_1 registers and sends the valid hashID.
SDL assigns HMILevel from "default_hmi" section of assigned policies to appID_1 (e.g., background).
BasicCommunication.OnAppRegistered {appID_1, resumeVrGrammars: true}
SDL waits during 3 sec giving a chanse to any other ap to resume.
After 3 sec SDL restores appID_1`s HMILevel (that is, FULL)
BasicCommunication.ActivateApp {appID_1}
App_1 is activated to FULL
BasicCommunication.ActivateApp {code: 0}

NOTE

BasicCommunication.ActivateApp is used differently than the previously described SDL.ActivateApp, but the two can be easily confused.

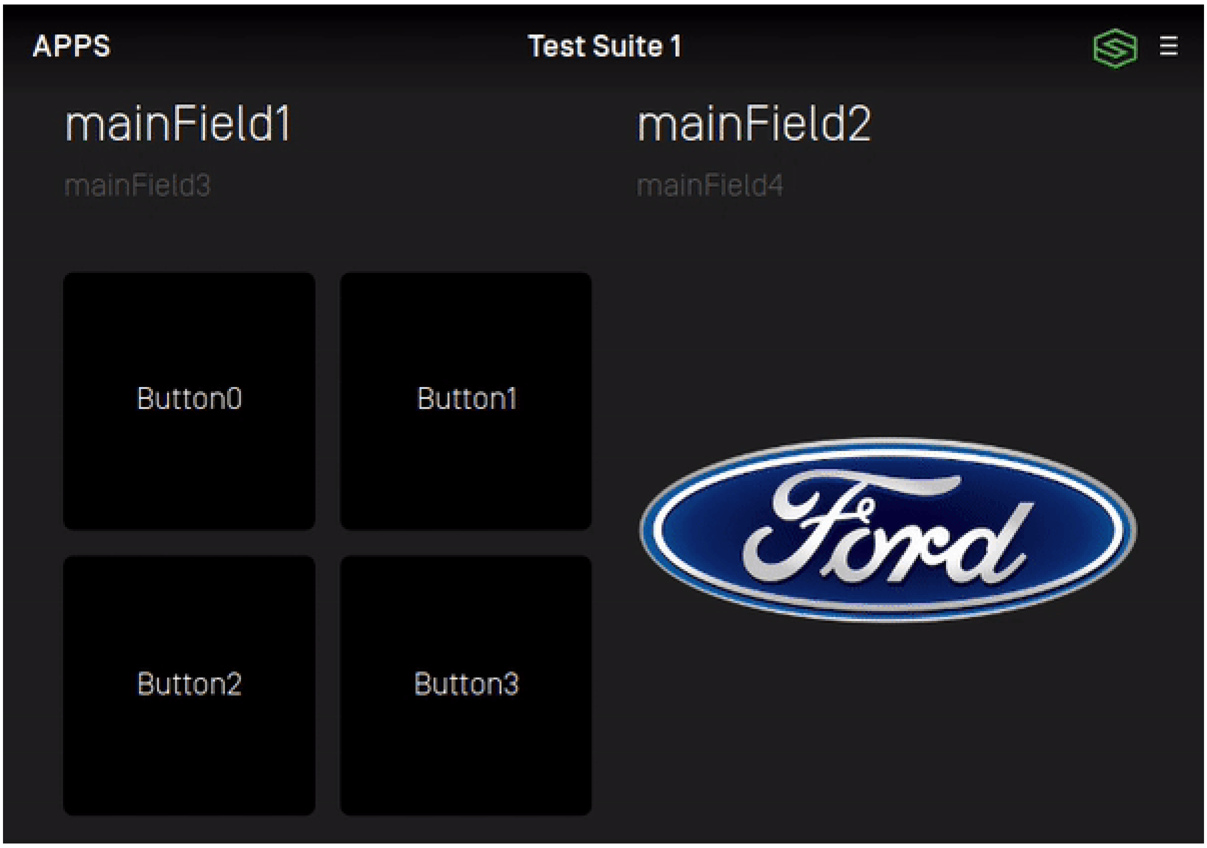
SDL.ActivateApp is a request originating from the HMI and should be sent when the user selects an app to activate.

BasicCommunication.ActivateApp is a request originating from SDL Core to move an app into a specific state. It is generally received by the HMI during app resumption.

Displaying Information

When an app wants to display information on the head unit, the HMI will receive a UI.Show request. The UI.Show request provides the HMI with the text, soft button information, and images an app has requested to display. The HMI should store the information in these requests for when an app is activated and put into full. UI.Show requests are not always sent when an app is activated and in view.

The following graphic shows what should happen when the HMI receives new text field and graphic information:



NOTE

The HMI should merge the information in `UI.Show` requests with existing show information received for an app.

For example, if the HMI receives a request with the text parameter `mainfield1` and a second request with the text parameter `mainfield2`, the HMI should display both `mainfield1` and `mainfield2`.

If an app wants to clear a text field that it sent in a previous `UI.Show` request, SDL Core will send the HMI a request with that parameter's value set to an empty string ("").

Media Layout Elements

Apps which use the `MEDIA` template have access to a few specific UI elements that are not available to non-media apps.

The following buttons can only be subscribed to by media apps and are generally only available in the `MEDIA` template layout:

- `PLAY_PAUSE`
- `SEEKLEFT`
- `SEEKRIGHT`
- `TUNEUP`
- `TUNEDOWN`

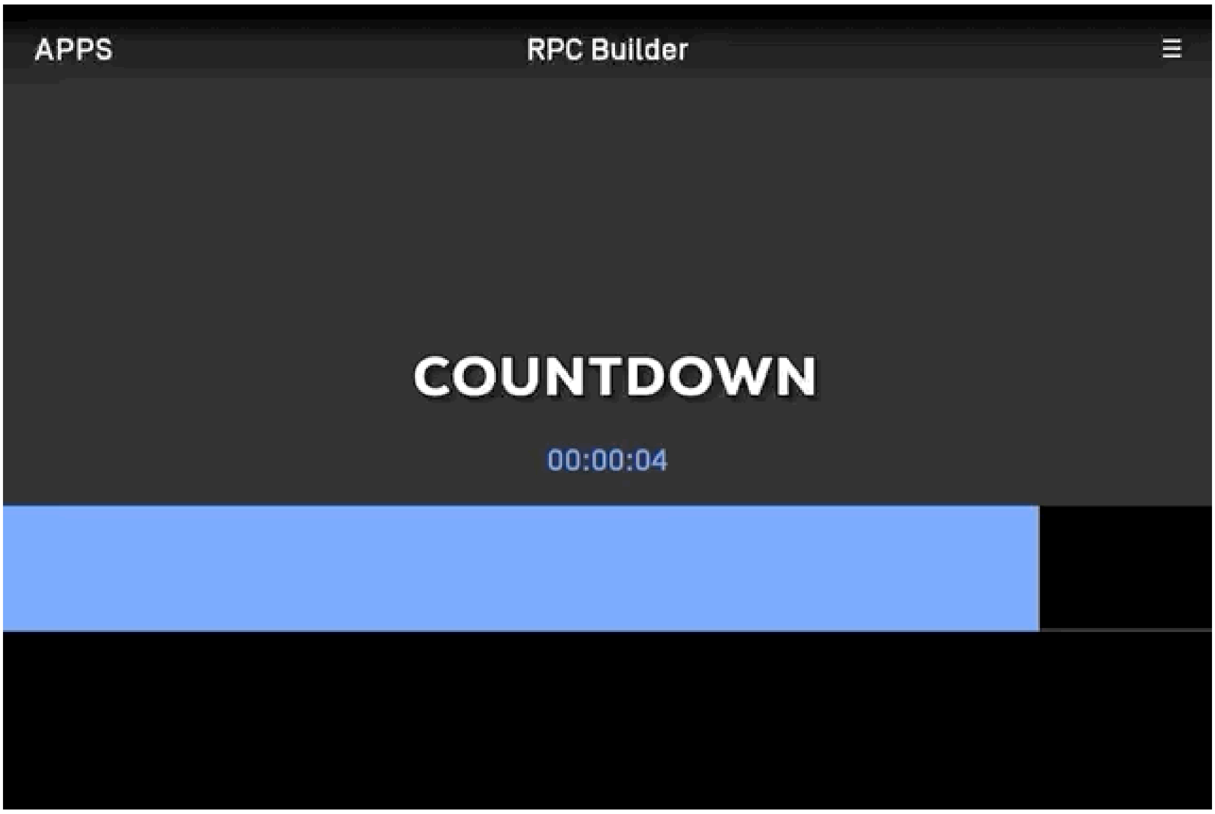
NOTE

Prior to RPC Spec version 5.0, the `OK` button name (which is available to all apps) was used by media apps for play/pause toggling.
With the release of version 5.0, the `PLAY_PAUSE` button name was introduced, allowing the HMI to have a separate `OK` and `PLAY_PAUSE` button for media apps.

Media apps have access to the media timer UI element via the `UI.SetMediaClockTimer` request. Similar to the `UI.Show` request, the HMI should keep track of the timer state for each app separately and display the appropriate state of the timer when the app is brought to the foreground. The HMI should react to the `UI.SetMediaClockTimer` request depending on the value of the `updateMode` parameter:

- `COUNTUP`: Begin counting up from `startTime` at the specified `countRate`, stopping at `endTime` if provided
- `COUNTDOWN`: Begin counting down from `startTime` at the specified `countRate`, stopping at `endTime` if provided
- `PAUSE`: Pause the existing timer at the current state, if running
- `RESUME`: Resume the previously paused timer starting from its paused state, counting at the specified `countRate`
- `CLEAR`: Clear the existing timer state, displaying the element as it was before the media timer was first set

The following graphic shows what should happen when the HMI receives a UI.SetMediaClockTimer request with each of these updateMode values:



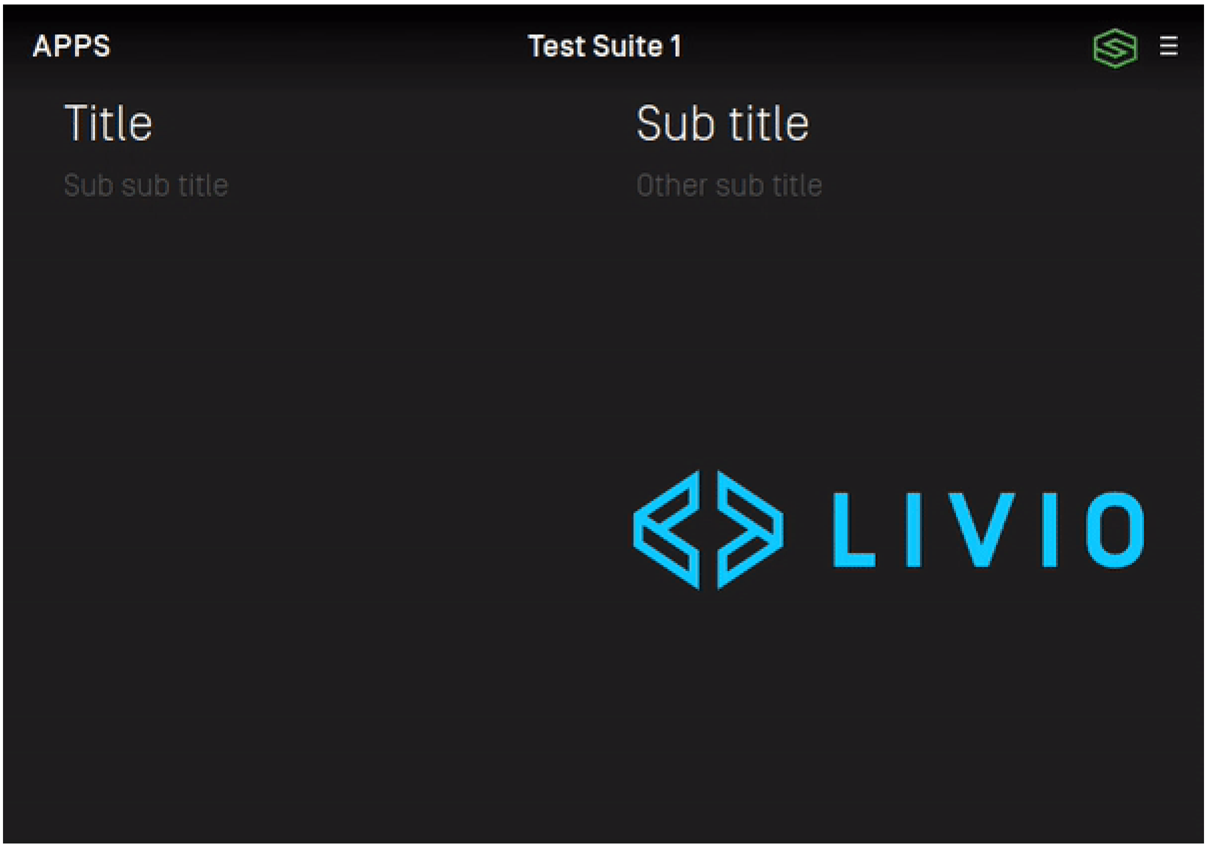
Implementing Soft Buttons

NOTE

As of Core 8.0.0 it is important to include CUSTOM_BUTTON in the response to Buttons.GetCapabilities so that an app may subscribe to soft buttons.

A Softbutton received from a UI.Show request should be displayed when the app is displaying a template. A template can have a max of 8 Softbuttons . These buttons can be of type `TEXT`, `IMAGE`, or `BOTH`.

The following graphic displays how `Softbuttons` in a `UI.Show` request can be displayed:



The HMI should keep an internal state of `SoftButtons` received by `UI.Show` requests, similar to how text fields and graphics are stored. Each `SoftButton` has a unique ID which must be saved by the HMI. These IDs are used in any messages sent to SDL Core when a user interacts with a `SoftButton`.

The actions expected of the HMI when the user selects a `SoftButton` are:

- HMI sends a notification `UI.OnButtonEvent` with buttonEventMode = `DOWN` when the user presses a button.
- HMI sends a notification `UI.OnButtonEvent` with buttonEventMode = `UP` when the user releases a button.
- HMI sends a notification `UI.OnButtonPress` with buttonPressMode = `SHORT` or `LONG`, depending on how long the user holds the button in a down state.

NOTE

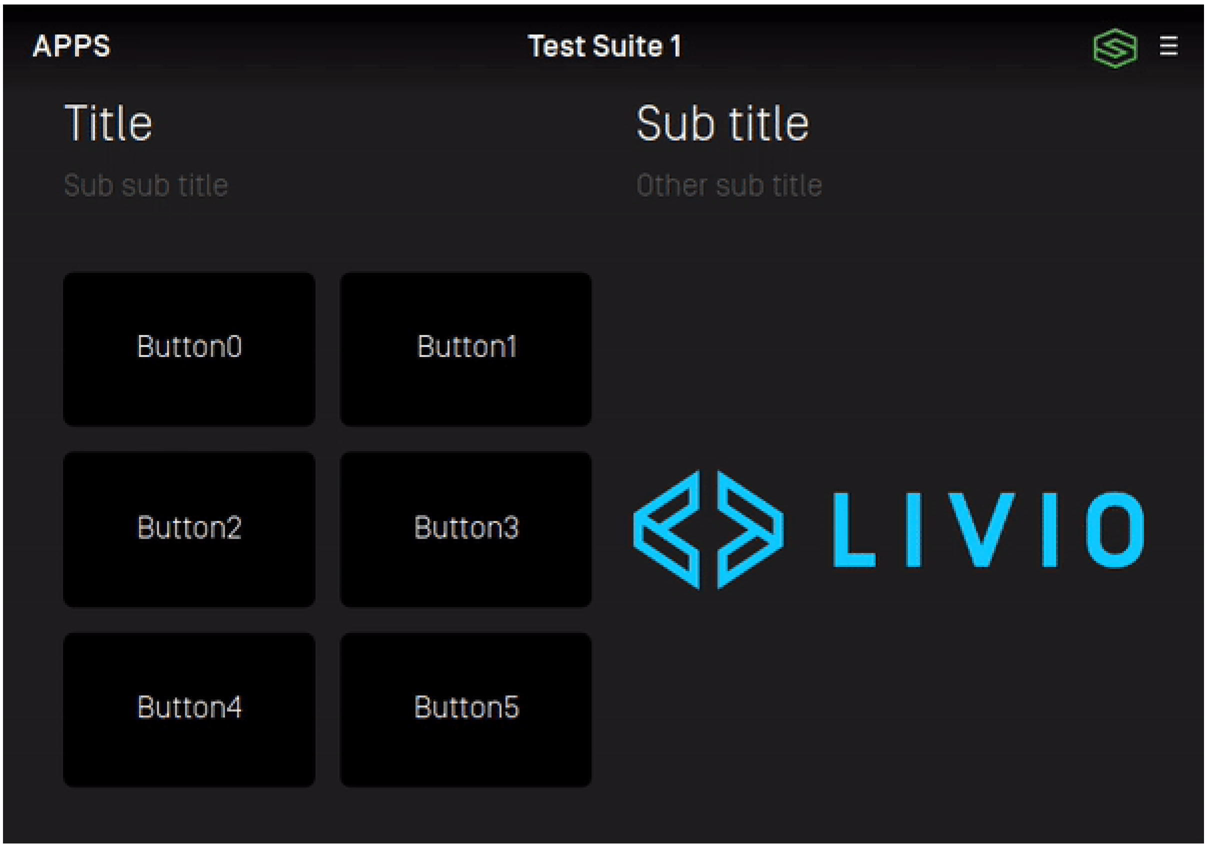
Not all HMIs support the ability to detect a button press duration, or differentiate between an up and down button event. In this case the HMI should make sure its ButtonCapabilities are accurately sent to Core via the `ButtonCapabilities` parameter in `UI.GetCapabilities`.

More on HMI capabilities.

Switching Templates

SDL Core can request the HMI to change an app's template using a `UI.Show` request.

The following graphic demonstrates switching templates while maintaining the same text, buttons, and graphic:



In order to specify the template to be displayed, the `UI.Show` request uses the `templateConfiguration` parameter, which includes a string for the requested layout.

Using `UI.Show` is the preferred method because the request can be used to change the layout of the screen and the screen contents in a single request. This helps prevent lag and screen flashing when an app wants to change an app template.

An SDL app should only request to view templates that are supported in the HMI Capabilities. The HMI may return a failed response to Core in the event an unsupported template is requested.

More on HMI capabilities.

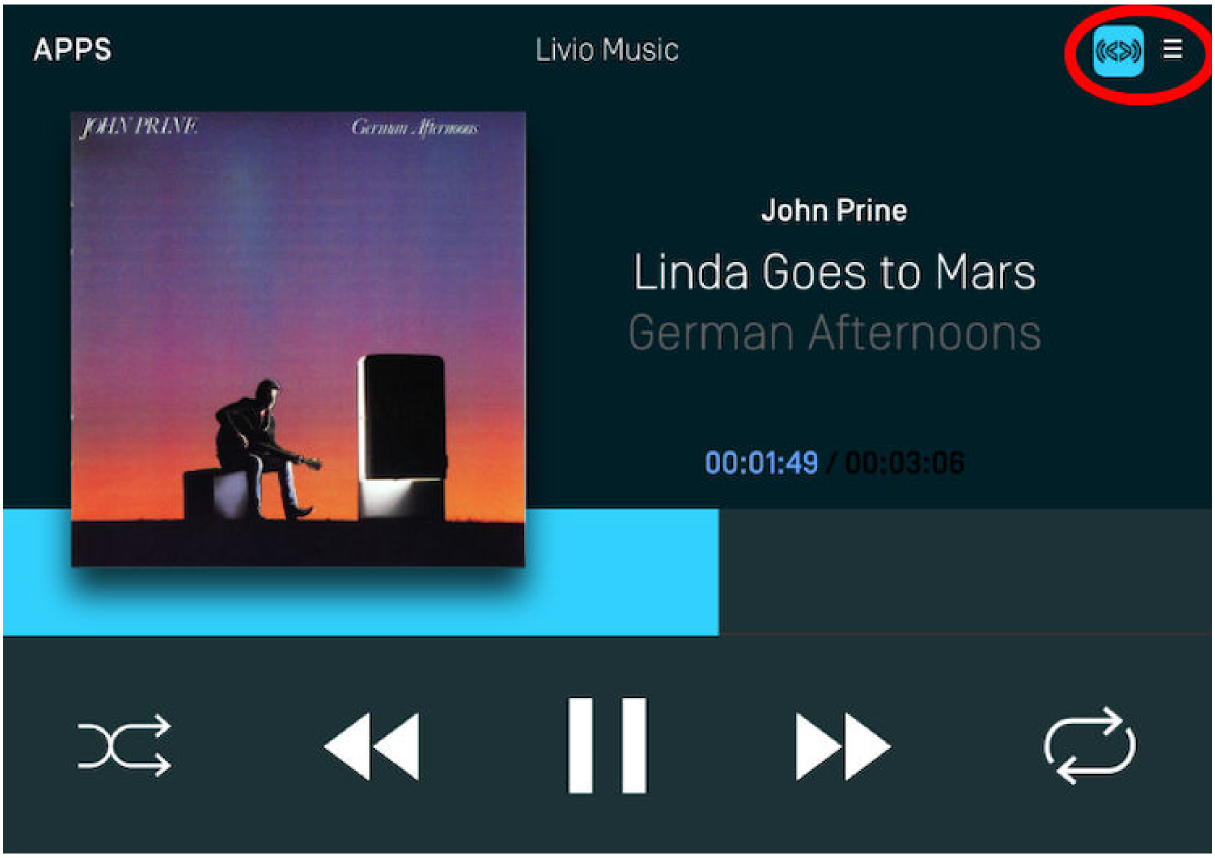
Supported Template Views

A reference list for all supported template views can be found here. This list shows screenshots of the 15 supported template views and how their text, graphic, and soft button components are arranged.

The defined strings for each template can be found in the PredefinedLayout enum in the Mobile API RPC Specification.

Creating the App Menu

Each application is able to maintain a list of menu commands through SDL. This in-app menu should be made accessible from an app's template view. Please note the example placement of the menu icon in the top right of the screenshot below.



NOTE

If the user chooses to open the menu, the HMI must send a `UI.OnSystemContext` notification with the `SystemContext` enum: `MENU`. After the user exits the menu, another `UI.OnSystemContext` notification must be sent with the `SystemContext` value: `MAIN`.

The contents of the app's menu are populated by the RPC `UI.AddCommand`. Each `UI.AddCommand` received corresponds to an individual menu item. When the user selects a menu item via the UI, the HMI should send a `UI.OnCommand` notification. It is best practice to exit the menu after a user makes a selection from the list of commands.

NOTE

Several menu items can have the same menuName. It is the app developer's responsibility to make commands clear to the user and not confusing in the case that several commands are given the same name.

There are some minor customization options available for the app menu. An HMI can choose to implement the app menu in a tile view, list view, or both. If an app has a preference for a type of menu layout, the HMI will receive a `UI.SetGlobalProperties` request from SDL Core containing this preference in the `menuLayout` field.

SDL also supports nested submenus which can be created using the RPC `UI.AddSubMenu`. If this request does not contain a `parentID` parameter (or `parentID` is 0) then the submenu should be made accessible by the top level menu. If the request contains a `parentID`, the new submenu should be added as an item to the submenu who's `menuID` matches the incoming `parentID`.

Menu commands that are populated by `UI.AddCommand` with a `parentID` value should be added as a menu item to the submenu who's `menuID` matches the incoming `parentI`

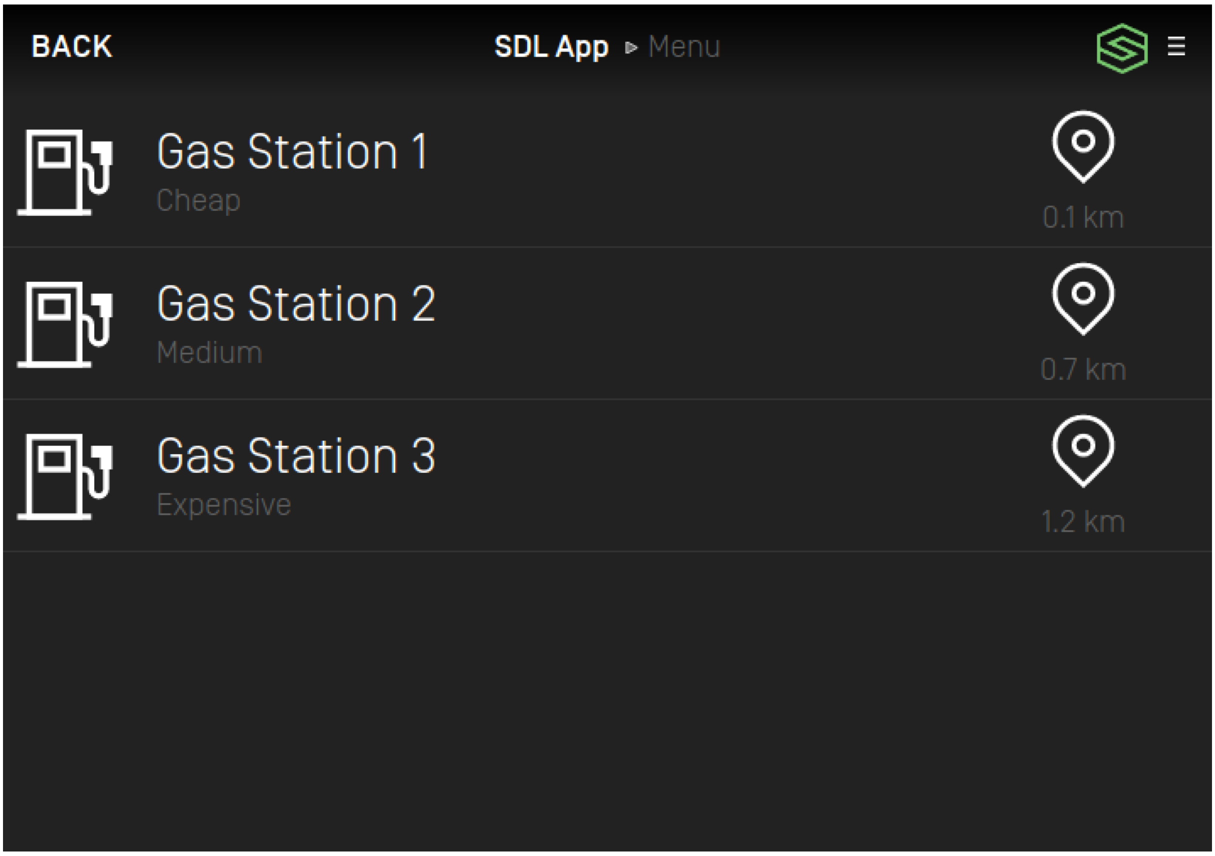
`D`.

NOTE

If the HMI is in a driver distraction mode, the HMI should show a maximum of "N" menu items in a given menu, as defined by the value of the `menuLength` driver distraction capability.

If the HMI is in a driver distraction mode, the HMI must restrict the user from accessing nested submenus beyond "N" levels deep, as defined by the value of the `subMenuDepth` driver distraction capability.

LIST MENU EXAMPLE

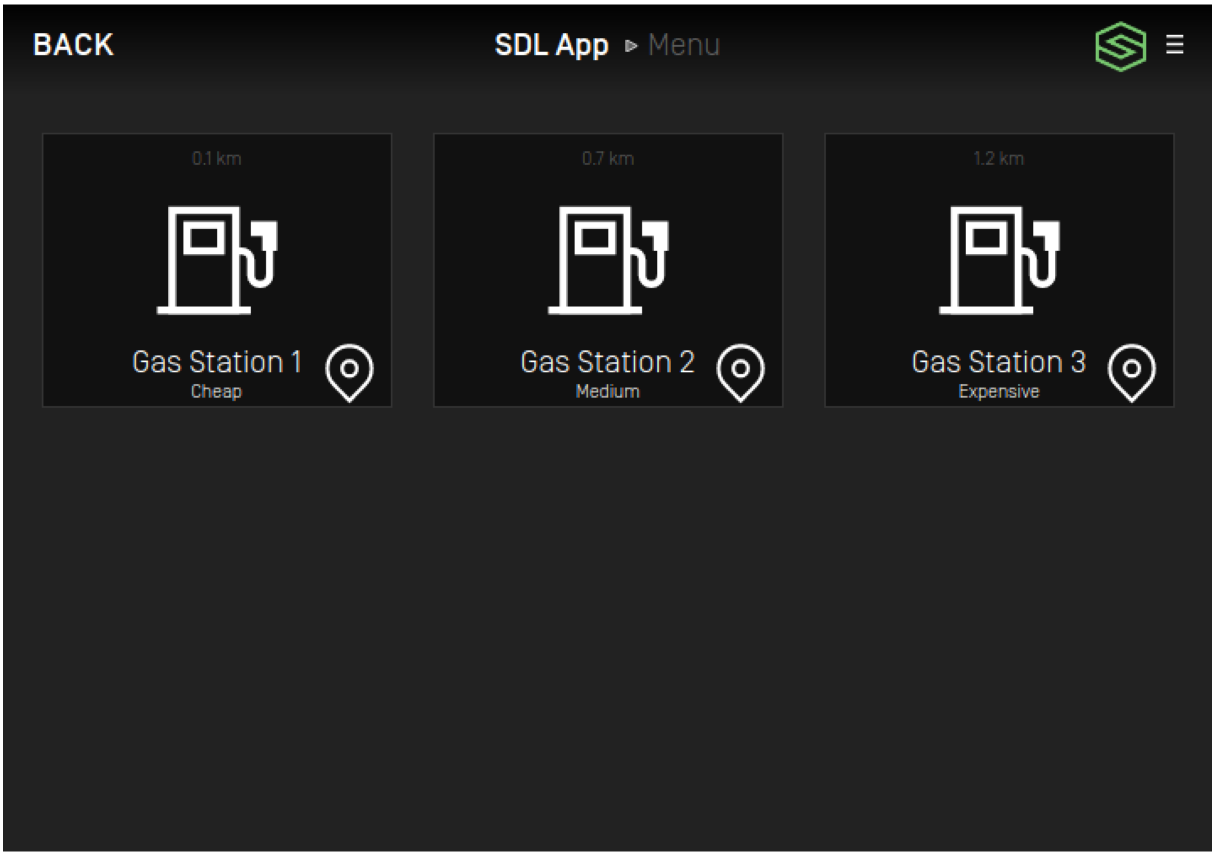


Gas Station 1
Cheap
0.1 km

Gas Station 2
Medium
0.7 km

Gas Station 3
Expensive
1.2 km

TILES MENU EXAMPLE



Dynamic Menu Updating

SDL enables the ability to dynamically load menu items and icons to improve system performance. In some cases an app may submit a large number of menu commands, sub menus, and icons. Processing these assets can use a large amount of system resources. To help mitigate performance issues, the HMI can choose when to request resources, at which time the app can update SDL Core with the missing menu contents.

`UI.OnUpdateFile` is used to request missing menu icons, and `UI.OnUpdateSubMenu` is used to request missing sub menu contents. These notifications can be sent to SDL Core when the user is in close proximity to the menu items. For example, if the user opens a menu that contains a list of submenus, the HMI may then request those submenus are populated via `AddCommand` requests from mobile. Additionally, if the HMI implements a paginated menu, the HMI may request all icons for the menu items that are on the next page.

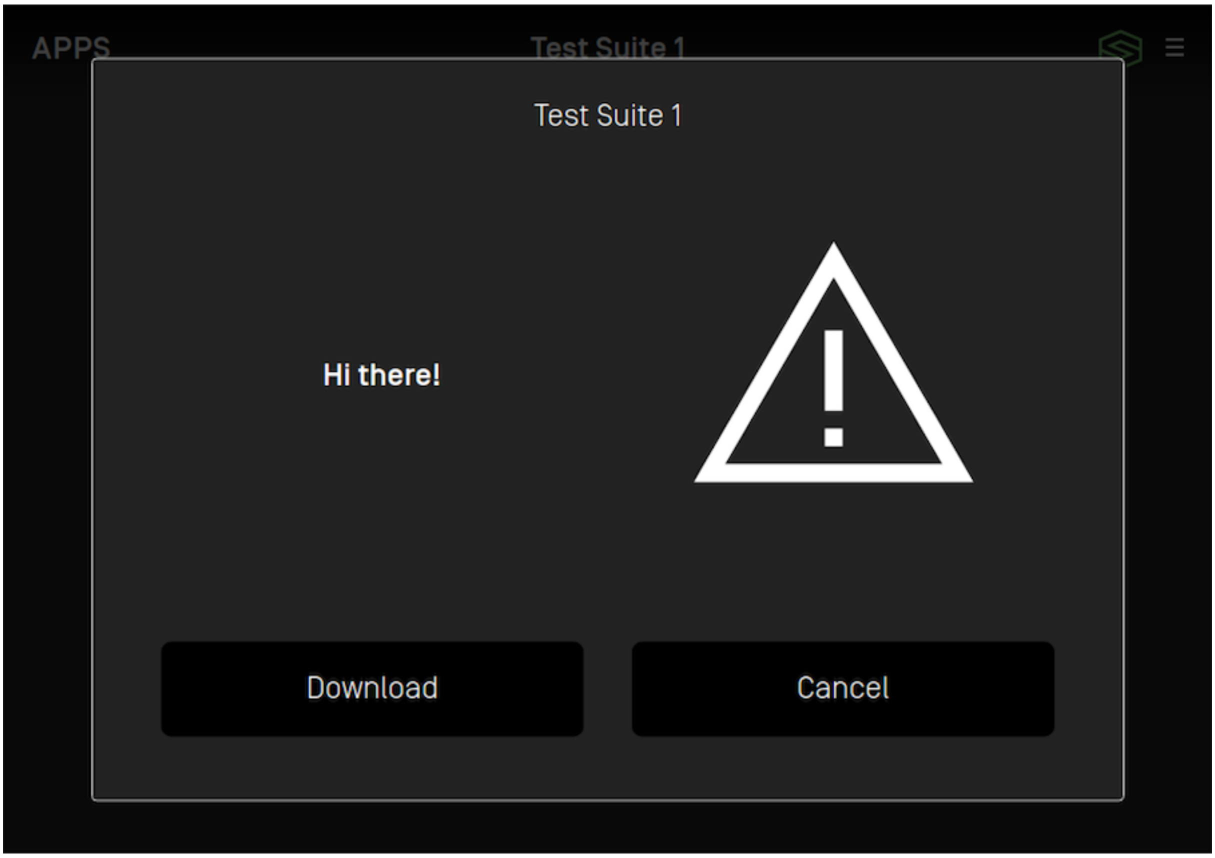
The HMI is free to manage these resources and delete them in case of memory issues. If the commands or icons are needed in the future, the HMI can send the appropriate notifications to request the menu contents be updated repeatedly from the SDL application.

Implementing Popups

There are several RPCs which are used to display a popup or an overlay to the user.

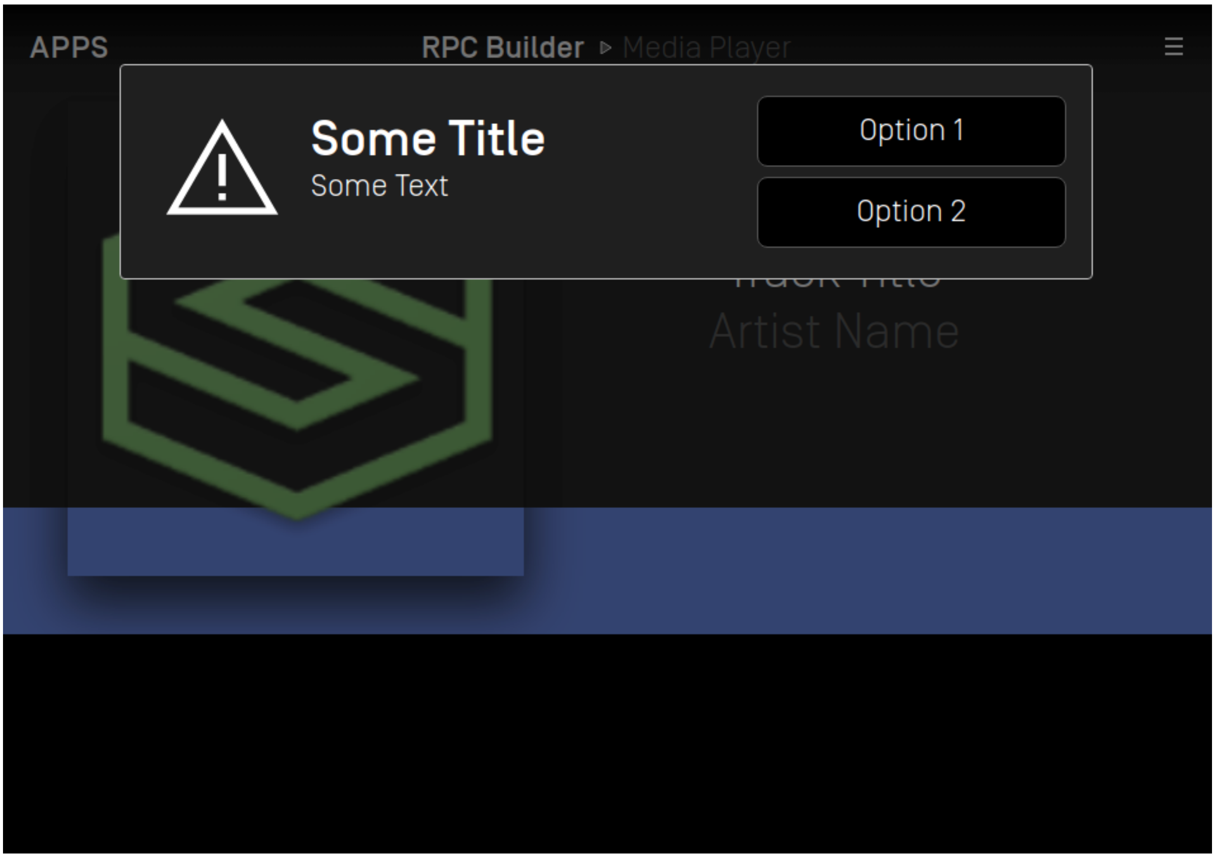
UI.Alert

`Alert` is used to display a simple popup that can contain an image, text, and buttons.



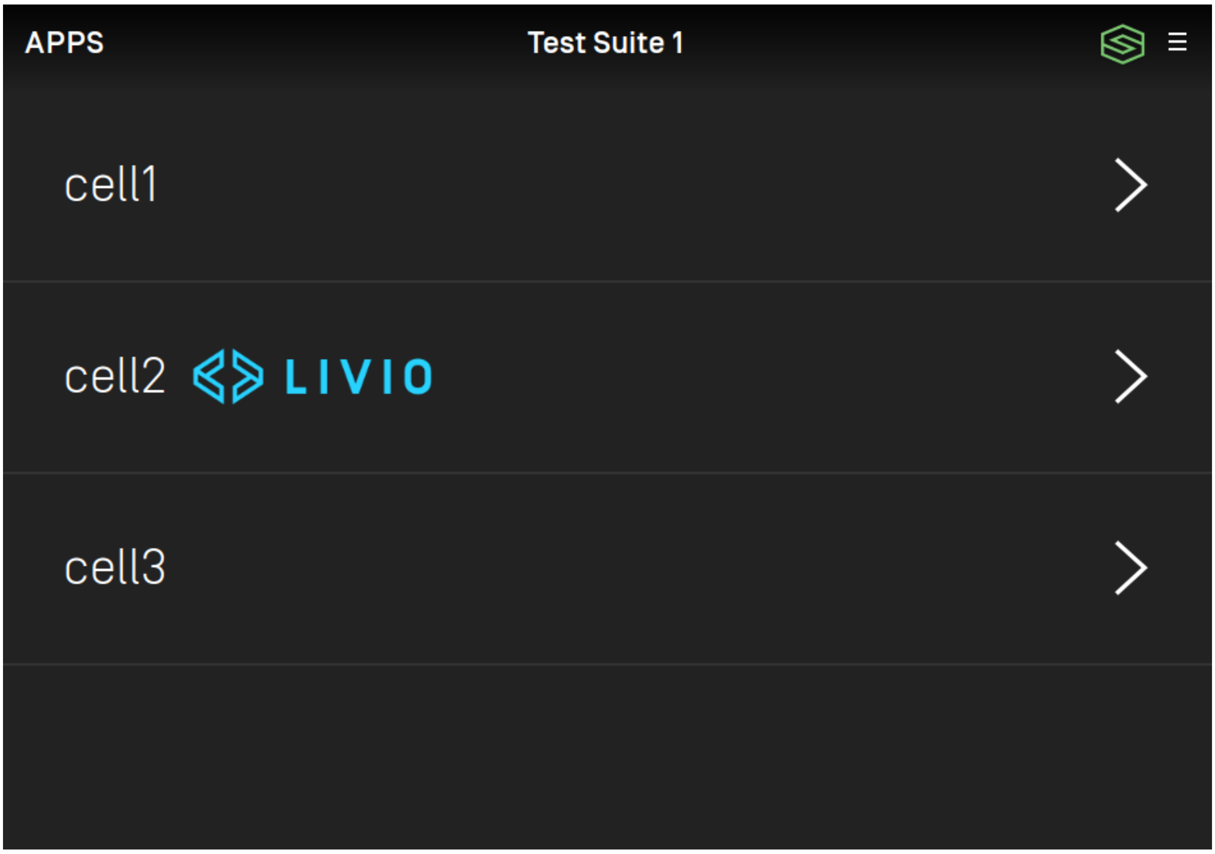
UI.SubtleAlert

`SubtleAlert` is used to display a notification-style popup that can contain an image, text, and buttons.



UI.PerformInteraction

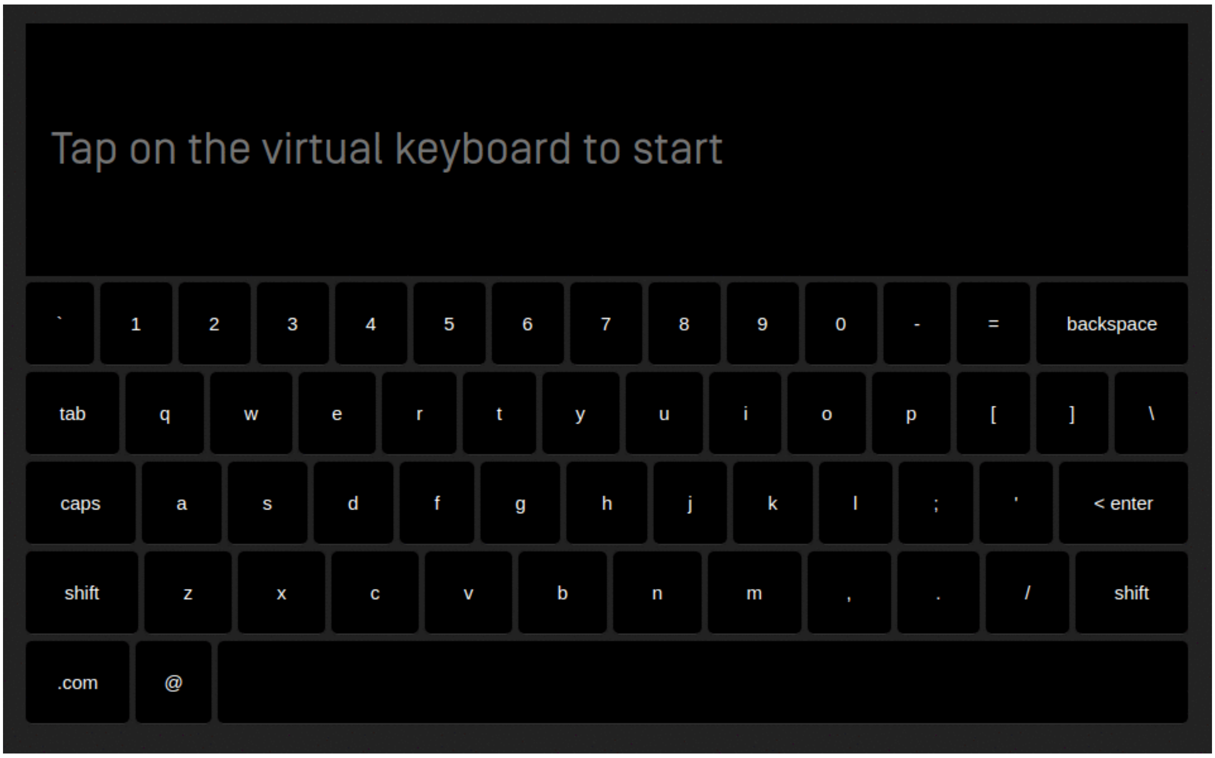
`PerformInteraction` is used to display a popup with contents which are displayed in a similar way to the app menu.



`PerformInteraction` has multiple layout types, including an on screen keyboard. SDL currently supports the following keyboard layouts:

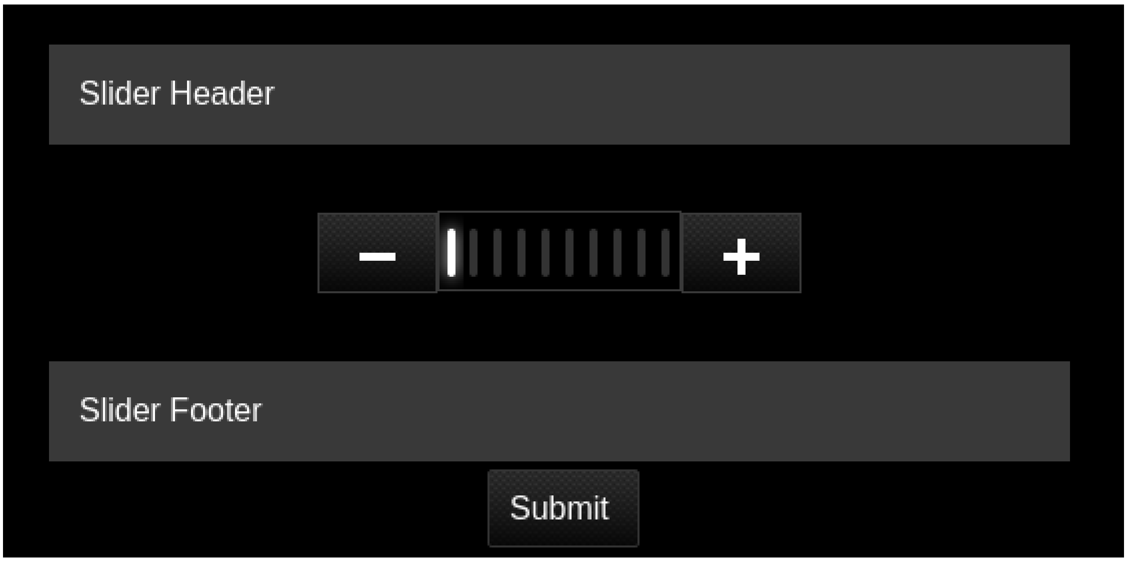
- QWERTY
- QWERTZ
- AZERTY
- NUMERIC

The on screen keyboard may be configured by the app to allow masking inputs and allow an app to configure special characters. These keyboard capabilities are optional and the HMI should communicate its capabilities to SDL Core via the `KeyboardCapabilities` Struct.


Tap on the virtual keyboard to start
` 1 2 3 4 5 6 7 8 9 0 - = backspace
tab q w e r t y u i o p [ ] \
caps a s d f g h j k l ; ' < enter
shift z x c v b n m , . / shift
.com @

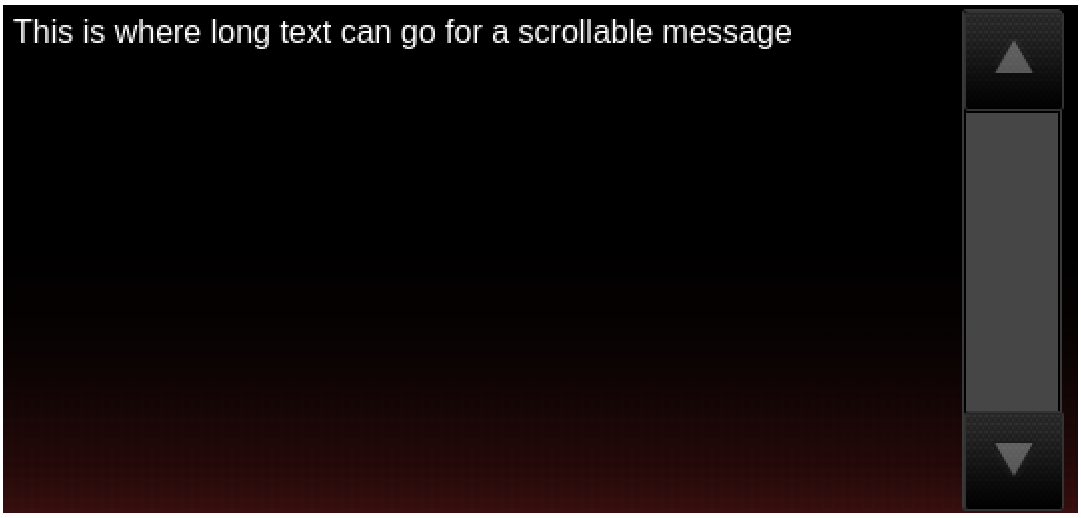
UI.Slider

`Slider` is used to display a popup that allows the user to enter a value via a slider input.



UI.ScrollableMessage

`ScrollableMessage` is used to display a popup which shows a long message to the user that requires scrolling.



This is where long text can go for a scrollable message

NOTE

It is important that the HMI sends SDL Core a `UI.OnSystemContext` notification when displaying and closing a popup. A `systemContext` value of `ALERT` is used when `UI.Alert` or `UI.SubtleAlert` is active, `HMI_OBSCURED` is used for all other popups.

Navigating Through the IVI

It is common for an SDL UI to be integrated into an existing OEM's UI. In order for SDL Core to work well with a head unit that has other embedded components, the HMI should make use of the `BasicCommunication.OnEventChanged` notification. This notification allows connected SDL applications to receive updates about their HMI status when a user interacts with other components like the embedded navigation or radio.

For example, if an SDL media application is active and is playing audio, then the user switches the audio source to the embedded radio, the HMI should send SDL Core a `BasicCommunication.OnEventChanged` notification with `eventName = AUDIO_SOURCE` and `isActive = true`. This HMI notification will let the media application know that it no longer has control of the audio source.

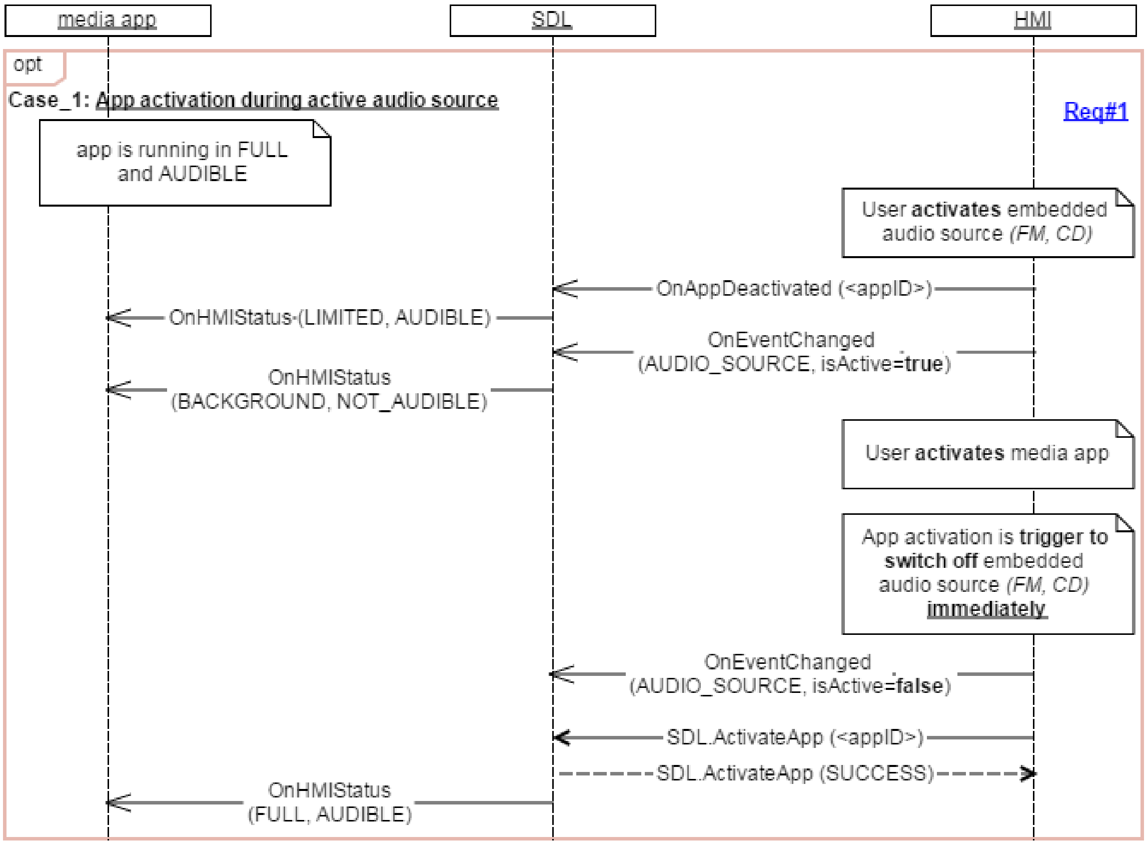
If the user selects the media app as the audio source again, the HMI should send the same `BasicCommunication.OnEventChanged` notification, but with `isActive = false`. This will indicate to SDL Core that the application has regained control of the audio.

The following gif and sequence diagram demonstrate the behavior of switching between an SDL media app and the embedded IVI audio.

SEQUENCE DIAGRAM

OnEventChanged Sequence Diagram

View Diagram


media app
SDL
HMI
opt
Case_1: App activation during active audio source
Req#1
app is running in FULL and AUDIBLE
User activates embedded audio source (FM, CD)
OnAppDeactivated (<appID>)
OnHMIStatus (LIMITED, AUDIBLE)
OnEventChanged (AUDIO_SOURCE, isActive=true)
OnHMIStatus (BACKGROUND, NOT_AUDIBLE)
User activates media app
App activation is trigger to switch off embedded audio source (FM, CD) immediately
OnEventChanged (AUDIO_SOURCE, isActive=false)
SDL.ActivateApp (<appID>)
SDL.ActivateApp (SUCCESS)
OnHMIStatus (FULL, AUDIBLE)

Defining the UI Capabilities

There are several ways that the HMI should communicate its UI capabilities to SDL Core. When first connecting the HMI to SDL Core, SDL Core will send a `UI.GetCapabilities` request (a similar GetCapabilities request is sent for every interface). The HMI's response should include accurate information relating to its supported display capabilities, audio pass through capabilities, soft button capabilities, and various other system capabilities (See UI.GetCapabilities).

It is likely that the UI capabilities will be different for each template view, therefore it is important for the HMI to send updates about its capabilities to SDL Core. For example, if an app requests a new template configuration, after switching to that view the HMI must send an `OnSystemCapabilityUpdated` notification for `"systemCapabilityType": "DISPLAYS"`.

As an example, if SDL Core requests to change the layout to the `MEDIA` template, the `OnSystemCapabilitiesUpdated` notification parameters may look something like this (taken from the Generic HMI):

```
{
  "appID":"1234",
  "systemCapability":{
    "systemCapabilityType":"DISPLAYS",
    "displayCapabilities":{
      "displayName":"GENERIC_HMI",
      "windowTypeSupported":[
        {
          "type":"MAIN",
          "maximumNumberOfWindows":1
        }
      ],
      "windowCapabilities":[
        {
          "windowID":0,
          "textFields":[
            {
              "name":"mainField1",
              "characterSet":"UTF_8",
              "width":500,
              "rows":1
            },
            {
              "name":"mainField2",
              "characterSet":"UTF_8",
              "width":500,
              "rows":1
            },
            {
              "name":"mainField3",
              "characterSet":"UTF_8",
              "width":500,
              "rows":1
            },
            {
              "name":"statusBar",
              "characterSet":"UTF_8",
              "width":500,
              "rows":1
            },
            {
              "name":"mediaClock",
              "characterSet":"UTF_8",
              "width":500,
              "rows":1
            },
            {
              "name":"mediaTrack",
              "characterSet":"UTF_8",
              "width":500,
              "rows":1
            },
      {
        "name":"templateTitle",
        "characterSet":"UTF_8",
        "width":50,
        "rows":1
      },
      {
        "name":"alertText1",
        "characterSet":"UTF_8",
        "width":500,
        "rows":1
      },
      {
        "name":"alertText2",
        "characterSet":"UTF_8",
        "width":500,
        "rows":1
      },
      {
        "name":"alertText3",
        "characterSet":"UTF_8",
        "width":500,
        "rows":1
      },
      {
        "name":"subtleAlertText1",
        "characterSet":"TYPE2SET",
        "width":500,
        "rows":1
      },
      {
        "name":"subtleAlertText2",
        "characterSet":"TYPE2SET",
        "width":500,
        "rows":1
      },
      {
        "name":"subtleAlertSoftButtonText",
        "characterSet":"TYPE2SET",
        "width":50,
        "rows":1
      },
      {
        "name":"menuName",
        "characterSet":"UTF_8",
        "width":500,
        "rows":1
      },
      {
        "name":"secondaryText",
        "characterSet":"UTF_8",
        "width":500,
        "rows":1
      },
        {
          "name":"tertiaryText",
          "characterSet":"UTF_8",
          "width":500,
          "rows":1
        },
        {
          "name":"menuTitle",
          "characterSet":"UTF_8",
          "width":500,
          "rows":1
        },
        {
          "name": "menuCommandSecondaryText",
          "characterSet": "UTF_8",
          "width": 500,
          "rows": 1
        },
        {
          "name": "menuCommandTertiaryText",
          "characterSet": "UTF_8",
          "width": 500,
          "rows": 1
        },
        {
          "name": "menuSubMenuSecondaryText",
          "characterSet": "UTF_8",
          "width": 500,
          "rows": 1
        },
        {
          "name": "menuSubMenuTertiaryText",
          "characterSet": "UTF_8",
          "width": 500,
          "rows": 1
        }
      ],
      "imageFields":[
        {
          "name":"choiceImage",
          "imageTypeSupported":[
            "GRAPHIC_PNG"
          ],
          "imageResolution":{
            "resolutionWidth":40,
            "resolutionHeight":40
          }
        },
        {
          "name":"softButtonImage",
          "imageTypeSupported":[
            "GRAPHIC_PNG"
          ],
          "imageResolution":{
```

```
          "resolutionWidth":50,
          "resolutionHeight":50
        }
      },
      {
        "name":"softButtonImage",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":50,
          "resolutionHeight":50
        }
      },
      {
        "name":"menuIcon",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":40,
          "resolutionHeight":40
        }
      },
      {
        "name":"cmdIcon",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":150,
          "resolutionHeight":150
        }
      },
      {
        "name":"appIcon",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":50,
          "resolutionHeight":50
        }
      },
      {
        "name":"graphic",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":360,
          "resolutionHeight":360
        }
      },
      {
        "name":"alertIcon",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":225,
          "resolutionHeight":225
        }
      },
      {
        "name":"subtleAlertIcon",
        "imageTypeSupported":[
          "GRAPHIC_PNG"
        ],
        "imageResolution":{
          "resolutionWidth":40,
          "resolutionHeight":40
        }
      }
      {
        "name": "menuCommandSecondaryImage",
        "imageTypeSupported": [
          "GRAPHIC_BMP",
          "GRAPHIC_JPEG",
          "GRAPHIC_PNG"
        ],
        "imageResolution": {
          "resolutionWidth": 65,
          "resolutionHeight": 65
        }
      },
      {
        "name": "menuSubMenuSecondaryImage",
        "imageTypeSupported": [
          "GRAPHIC_BMP",
          "GRAPHIC_JPEG",
          "GRAPHIC_PNG"
        ],
        "imageResolution": {
          "resolutionWidth": 65,
          "resolutionHeight": 65
        }
      }
    ],
    "imageTypeSupported":[
      "DYNAMIC",
      "STATIC"
    ],
    "templatesAvailable":[
      "DEFAULT",
      "MEDIA",
      "NON-MEDIA",
      "LARGE_GRAPHIC_WITH_SOFTBUTTONS",
        "LARGE_GRAPHIC_ONLY",
        "GRAPHIC_WITH_TEXTBUTTONS",
        "TEXTBUTTONS_WITH_GRAPHIC",
        "TEXTBUTTONS_ONLY",
        "TEXT_WITH_GRAPHIC",
        "GRAPHIC_WITH_TEXT",
        "DOUBLE_GRAPHIC_WITH_SOFTBUTTONS"
      ],
      "buttonCapabilities":[
        {
          "shortPressAvailable":true,
          "longPressAvailable":false,
          "upDownAvailable":false,
          "name":"OK"
        },
        {
          "shortPressAvailable":true,
          "longPressAvailable":false,
          "upDownAvailable":false,
          "name":"PLAY_PAUSE"
        },
        {
          "shortPressAvailable":true,
          "longPressAvailable":false,
          "upDownAvailable":false,
          "name":"SEEKLEFT"
        },
        {
          "shortPressAvailable":true,
          "longPressAvailable":false,
          "upDownAvailable":false,
          "name":"SEEKRIGHT"
        }
      ],
      "softButtonsCapabilities":[
        {
          "shortPressAvailable":true,
          "longPressAvailable":false,
          "upDownAvailable":false,
          "imageSupported":true,
          "textSupported":true
        },
        {
          "shortPressAvailable":true,
          "longPressAvailable":false,
          "upDownAvailable":false,
          "imageSupported":true,
          "textSupported":true
        }
      ],
      "menuLayoutsAvailable":[
        "LIST",
        "TILES"
      ],
        "keyboardCapabilities": {
          "maskInputCharactersSupported": true,
          "supportedKeyboards": [
            {
              "keyboardLayout": "QWERTY",
              "numConfigurableKeys": 10
            },
            {
              "keyboardLayout": "QWERTZ",
              "numConfigurableKeys": 10
            },
            {
              "keyboardLayout": "AZERTY",
              "numConfigurableKeys": 10
            },
            {
              "keyboardLayout": "NUMERIC",
              "numConfigurableKeys": 0
            }
          ]
        }
      }
    ]
  }
}
}
```

Vehicle Data

The purpose of this guide is to explain how vehicle data items can be exposed to app developers through the HMI.

Vehicle data can be exposed to app developers by creating a `VehicleInfo` component within your HMI. To communicate with this component, you will first need to register it with the message broker and respond to the `VehicleInfo.IsReady` message from SDL (see the Component Readiness Requests section for more information).

RPCs

Below are descriptions for the primary RPCs used by the `VehicleInfo` component of SDL. More information regarding this component is available in the `VehicleInfo` section of the HMI Documentation.

VehicleInfo.GetVehicleData

Description:

A request from Core to retrieve specific vehicle data items from the system.

Example Request:

```
{
    "id": 123,
    "jsonrpc": "2.0",
    "method": "VehicleInfo.GetVehicleData",
    "params" : {
        "speed" : true
    }
}
```

Example Response:

```
{
    "id": 123,
    "jsonrpc": "2.0",
    "result" : {
        "speed" : 100
    }
}
```

VehicleInfo.SubscribeVehicleData

Description:

A request from Core to receive periodic updates for specific vehicle data items from the system.

Example Request:

```
{
    "id": 123,
    "jsonrpc": "2.0",
    "method": "VehicleInfo.SubscribeVehicleData",
    "params" : {
        "speed" : true
    }
}
```

Example Response:

```
{
    "id": 123,
    "jsonrpc": "2.0",
    "result" : {
        "speed" : {
            "dataType" : "VEHICLEDATA_SPEED",
            "resultCode" : "SUCCESS"
        }
    }
}
```

VehicleInfo.UnsubscribeVehicleData

Description:

A request from Core to stop receiving periodic updates for specific vehicle data items from the system.

Example Request:

```
{
    "id": 123,
    "jsonrpc": "2.0",
    "method": "VehicleInfo.UnsubscribeVehicleData",
    "params" : {
        "speed" : true
    }
}
```

Example Response:

```
{
    "id": 123,
    "jsonrpc": "2.0",
    "result" : {
        "speed" : {
            "dataType" : "VEHICLEDATA_SPEED",
            "resultCode" : "SUCCESS"
        }
    }
}
```

VehicleInfo.OnVehicleData

Description:

A notification from the HMI indicating that one or more of the subscribed vehicle data items were updated.

Example Notification:

```
{
    "jsonrpc": "2.0",
    "method": "VehicleInfo.OnVehicleData",
    "result" : {
        "speed" : 100
    }
}
```

Available Vehicle Data Items

Below is a list of all of the vehicle data items which are available via SDL as of Release 6.1.0 of SDL Core. New vehicle data items are proposed regularly via the SDL Evolution process.

NAME	RESULT TYPE	DESCRIPTION
accPedalPosition	Float	Accelerator pedal position (as a number from 0 to 100 representing percentage depressed)
beltStatus	Common.BeltStatus	The status of each of the seat belts in the vehicle
bodyInformation	Common.BodyInformation	The body information for the vehicle, including information such as ignition status and door status
climateData	Common.ClimateData	Describes the climate status within the vehicle.
cloudAppVehicleID	String	Parameter used by cloud apps to identify a head unit
deviceStatus	Common.DeviceStatus	The device status, including information such as signal and battery strength
driverBraking	Common.VehicleDataEventStatus	The status of the brake pedal
electronicParkBrakeStatus	Common.ElectronicParkBrakeStatus	The status of the park brake as provided by Electric Park Brake (EPB) system
engineOilLife	Float	The estimated percentage of remaining oil life of the engine

NAME	RESULT TYPE	DESCRIPTION
engineTorque	Float	Torque value for the engine (in N*m) on non-diesel variants
externalTemperature	Float	The external temperature in degrees celsius
fuelLevel_State	Common.ComponentVolumeStatus	The status value corresponding to the general fuel level in the tank
fuelLevel	Float	The fuel level in the tank (as a percentage value)
fuelRange	Common.FuelRange Array	The estimate range in KM the vehicle can travel based on fuel level and consumption. Contains information on all fuel sources available to the vehicle (eg. GASOLINE and BATTERY for hybrid vehicles).
gearStatus	Common.GearStatus	The current status of the gear shifter.
gps	Common.GPSData	Location data from the onboard GPS in the vehicle
handsOffSteering	Boolean	Indicates whether the driver's hands are off the steering wheel.

NAME	RESULT TYPE	DESCRIPTION
headLampStatus	Common.HeadLampStatus	The current status of each of the head lamps
instantFuelConsumption	Float	The instantaneous fuel consumption of the vehicle in microlitres
odometer	Integer	The odometer value in kilometers
prndl	Common.PRNDL	The current status of the gear shifter. This parameter is deprecated and it is now covered in `gearStatus`.
rpm	Integer	The number of revolutions per minute of the engine
seatOccupancy	Common.SeatOccupancy	Describes the occupancy, belted status, and location for each seat in the vehicle.
speed	Float	The vehicle speed in kilometers per hour
stabilityControlsStatus	Common.StabilityControlsStatus	Describes the ignition switch stability.
steeringWheelAngle	Float	The current angle of the steering wheel (in degrees)
tirePressure	Common.TireStatus	Status information for each of the vehicle's tires

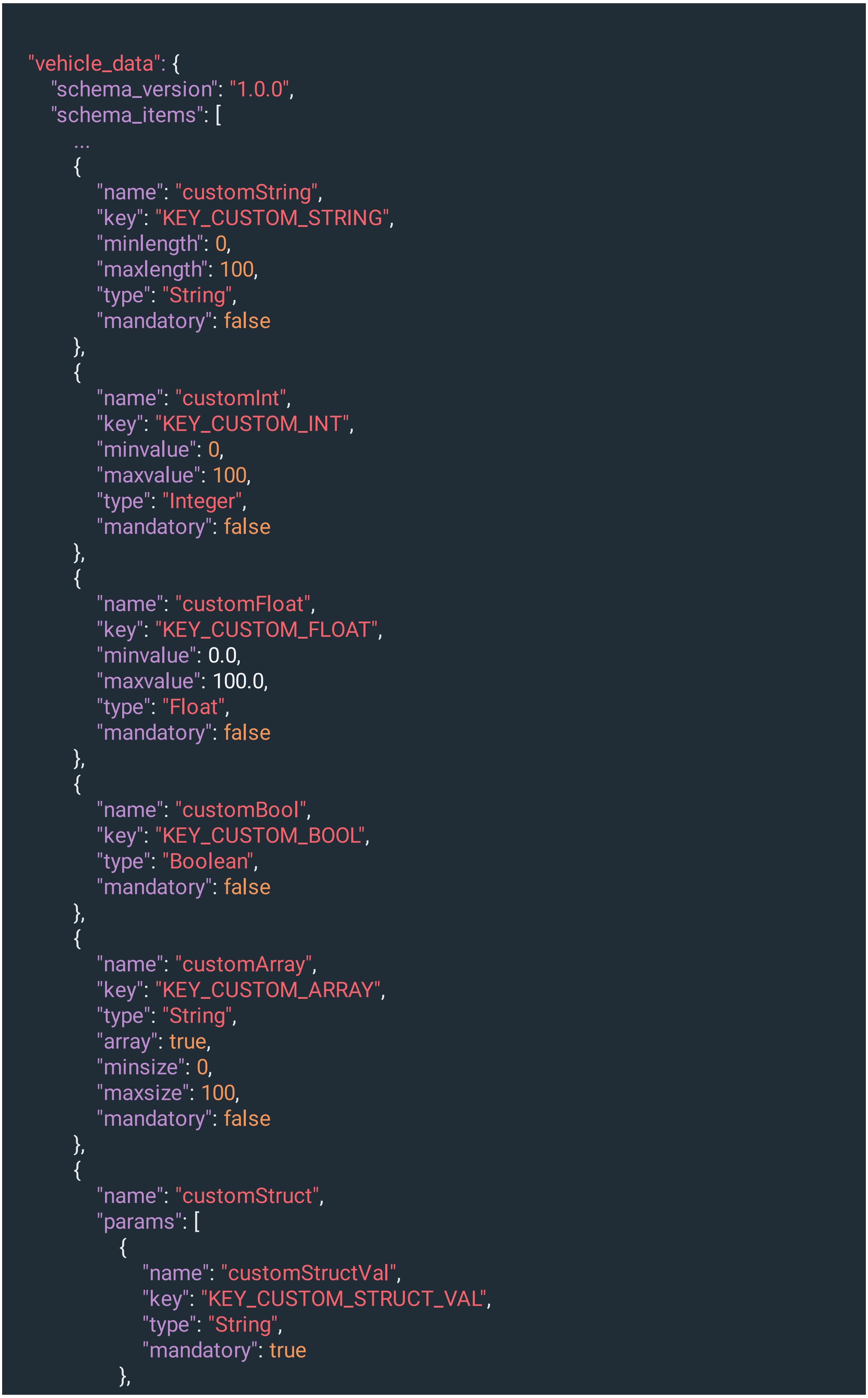
NAME	RESULT TYPE	DESCRIPTION
turnSignal	Common.TurnSignal	The current state of the turn signal indicator
vin	String	Vehicle identification number
windowStatus	Common.WindowStatus Array	Describes the status of each window for each door/liftgate etc.
wiperStatus	Common.WiperStatus	The current status of the wipers

Custom Vehicle Data Items

Starting with SDL Core version 6.0.0, custom vehicle data items can be defined via the policy table. See SDL-0173 for the full proposal details. These items are structured in a similar manner to the Mobile API and contained in the `vehicle_data` section of the policy table.

In addition to custom items, this feature can be used to expose other vehicle data items that were introduced to the project in later versions. This can be useful when the software version on the head unit cannot be updated easily. If a vehicle data item is added into the project, the definition of this item will be included in the policy table by default. Any vehicle data items which are defined in Core's local Mobile API will be ignored from the policy table, but newer items will be interpreted as custom items. This allows apps to use these data items normally if they are exposed by the head unit, even when they were not initially supported.

Example Entry



```
"vehicle_data": {
    "schema_version": "1.0.0",
    "schema_items": [
        ...
        {
            "name": "customString",
            "key": "KEY_CUSTOM_STRING",
            "minlength": 0,
            "maxlength": 100,
            "type": "String",
            "mandatory": false
        },
        {
            "name": "customInt",
            "key": "KEY_CUSTOM_INT",
            "minvalue": 0,
            "maxvalue": 100,
            "type": "Integer",
            "mandatory": false
        },
        {
            "name": "customFloat",
            "key": "KEY_CUSTOM_FLOAT",
            "minvalue": 0.0,
            "maxvalue": 100.0,
            "type": "Float",
            "mandatory": false
        },
        {
            "name": "customBool",
            "key": "KEY_CUSTOM_BOOL",
            "type": "Boolean",
            "mandatory": false
        },
        {
            "name": "customArray",
            "key": "KEY_CUSTOM_ARRAY",
            "type": "String",
            "array": true,
            "minsize": 0,
            "maxsize": 100,
            "mandatory": false
        },
        {
            "name": "customStruct",
            "params": [
                {
                    "name": "customStructVal",
                    "key": "KEY_CUSTOM_STRUCT_VAL",
                    "type": "String",
                    "mandatory": true
                },
                {
                    "name": "customStructVal2",
                    "key": "KEY_CUSTOM_STRUCT_VAL2",
                    "minvalue": 0,
                    "maxvalue": 100,
                    "type": "Integer",
                    "mandatory": true
                },
                {
                    "name": "customDeprecatedVal",
                    "key": "KEY_CUSTOM_DEPRECATED_VAL",
                    "minvalue": 0,
                    "maxvalue": 100,
                    "type": "Integer",
                    "mandatory": true,
                    "until": "7.0"
                },
                {
                    "name": "customDeprecatedVal",
                    "key": "KEY_CUSTOM_DEPRECATED_VAL",
                    "minvalue": 0,
                    "maxvalue": 100,
                    "type": "Integer",
                    "mandatory": true,
                    "deprecated": true,
                    "since": "7.0"
                }
            ],
            "key": "KEY_CUSTOM_STRUCT",
            "type": "Struct",
            "mandatory": false
        }
    ]
}
```

Custom Data Fields

- *name* : Is the vehicle data item in question. e.g. gps, speed etc. SDL core would use this as the vehicle data param for requests from the app and to validate policies permissions.
- *type* : Is the return data type of the vehicle data item. It can either be a generic SDL data type (Integer, String, Float, Boolean, Struct) or an enumeration defined in Mobile API XML. For a vehicle data item that has sub-params, this would be Struct.
- *key* : Is a reference for the OEM Network Mapping table which defines signal attributes for this vehicle data items. OEMs may use this table to differentiate between various vehicle and SW configurations. SDL core will pass along this

reference to HMI, and then HMI would be responsible to resolve this reference using the Vehicle Data Mapping table (see Vehicle Data Mapping File).

- *array* : A boolean value used to specify if the vehicle data item/param response is an array, rather than a single value of the given *type*.
- *mandatory* : A boolean value used to specify if the vehicle data param is mandatory to be included in response for the overall vehicle data item.
- *params* : A recursive list of sub-params for a vehicle data item, see example above (`customStruct`) for structure definition.
- *since*, *until* : String values related to API versioning which are optional per vehicle data item.
- *removed*, *deprecated* : Boolean values related to API versioning which are optional per vehicle data item.
- *minvalue*, *maxvalue* : Integer/Float values which are used for controlling the bounds of number values (Integer, Float).
- *minsize*, *maxsize* : Integer values which are used for controlling the bounds of array values (where *array* is **true**).
- *minlength*, *maxlength* : Integer values which are used for controlling the bounds of String values.

NOTE

- *name* is required for top level vehicle data items while *type*, *key* & *mandatory* are required fields for vehicle data & sub-params. However *array* can be omitted, in which case *array* defaults to **false**.
- *Custom/OEM Specific* vehicle data parameters that are not a part of the rpc spec should not have any version related tags included (*since*, *until*, *removed*, *deprecated*). These vehicle data parameters would not be able to have the same versioning system as the rpc spec, since any version number supplied would not be the version associated with any known public rpc spec.

Custom Vehicle Data Requests

Custom vehicle data requests have a separate structure to normal vehicle data requests. While normal vehicle data items are requested using the key structure of `"<name>: true"`, custom items are constructed using the *key* field and can have a nested structure (when requesting *Struct* items). For example, when requesting all of the vehicle data items which are defined above, the HMI would receive the following message:

```
{
    "id" : 139,
    "jsonrpc" : "2.0",
    "method" : "VehicleInfo.GetVehicleData",
    "params" : {
        "KEY_CUSTOM_STRING": true,
        "KEY_CUSTOM_INT": true,
        "KEY_CUSTOM_FLOAT": true,
        "KEY_CUSTOM_BOOL": true,
        "KEY_CUSTOM_ARRAY": true,
        "KEY_CUSTOM_STRUCT": {
            "KEY_CUSTOM_STRUCT_VAL": true,
            "KEY_CUSTOM_STRUCT_VAL2": true,
            "KEY_CUSTOM_DEPRECATED_VAL": true
        }
    }
}
```

Vehicle Data Mapping File

Since these keys may not be immediately known by the HMI, a vehicle data mapping file can be used to connect these keys to actual readable values from the vehicle. The HMI primarily uses this file to convert CAN data values into an SDL-compatible format. The location where this file is hosted can be specified in the policy table in the `module_config.endpoints.custom_vehicle_data_mapping` field (see Policy Endpoints). The format of this file is OEM-defined.

EXAMPLE FORMAT

```
{
   "version":"0.0.1",
   "date":"01-01-2020",
   "vehicleDataTable": [
      {
         "CGEA1.3c":{
            "defaultPowertrain": {
               "vehicleData": [
               ]
            },
            "PHEV":{
               "vehicleData":[
                  {
                     "key":"OEM_REF_FUELLEVEL",
                     "type":"Integer",
                     "minFrequency":200,
                     "maxLatency":10,
                     "messageName":"Cluster_Info3",
                     "messageID":"0x434",
                     "signalName":"FuelLvl_Pc_Dsply",
                     "transportChannel":"HS3",
                     "resolution":0.109,
                     "offset":-5.2174
                  }
               ]
            }
         }
      }
   ]
}
```

NOTE

In order for the HMI to determine when this file needs to be updated, this file can be assigned a version via the `module_config.endpoint_properties.custom_vehicle_data_mapping.version` field. The HMI can retrieve this field using the SDL.GetPolicyConfigurationData RPC.

Reading Raw CAN Data

In addition to complex vehicle data items, the vehicle data mapping file can also be used to make some CAN values directly readable via a *String* value:

POLICY DEFINITION

```
{
    "name":"messageName",
    "type":"String",
    "key":"OEM_REF_MSG",
    "array":true,
    "mandatory":false,
    "since":"X.x",
    "maxsize":100,
    "params":[]
}
```

HMI RESPONSE

```
{
    "messageName": "AB 04 D1 9E 84 5C B8 22"
}
```

Initial Configuration

SDL Core Setup

Before continuing, follow the Install and Run Guide for SDL Core if you have not already done so.

HMI Setup

The Generic HMI and SDL HMI both support streaming audio and some video formats in the browser using ffmpeg to transcode the video to VP8 WEBM or audio to WAV. Instructions to install the required dependencies can be found in the HMI README:

- SDL HMI Dependencies
- Generic HMI Dependencies

Prior to starting the HMI, you will need to run the backend server component (`./deploy_server.sh` in the HMI directory) which handles the transcoding process.

> **NOTE**
>
> Once you start a video stream it will take a few seconds for the transcoding session to begin. Your video stream should appear in the browser within about 10 seconds.

To stream without ffmpeg transcoding, or to stream a format that ffmpeg does not support, you can forgo starting the backend server and use gstreamer to consume your audio/video stream.

GSTREAMER Setup

It is easier to determine which gstreamer video sink will work in your environment by testing with a static file. This can be done by downloading this file and trying the following command.

Common values for sink:

- ximagesink (x visual environment sink)
- xvimagesink (xv visual environment sink)
- cacasink (ascii art sink)

```
gst-launch-1.0 filesrc location=/path/to/h264/file ! decodebin ! videoconvert ! <sink>
sync=false
```

If you're streaming video over TCP, you can point gstreamer directly to your phone's stream using

```
gst-launch-1.0 tcpclientsrc host=<Device IP Address> port=3000 ! decodebin !
videoconvert ! <sink> sync=false
```

Pipe Streaming

Configuration (smartDeviceLink.ini)

In the Core build folder, open `bin/smartDeviceLink.ini` and ensure the following values are set:

```
VideoStreamConsumer = pipe
AudioStreamConsumer = pipe
```

GStreamer Commands

After you start SDL Core, cd into the bin/storage directory and there should be a file named "video_stream_pipe". Use the gst-launch command that worked for your environment and set file source to the video_stream_pipe file. You should see “setting pipeline to PAUSED” and “Pipeline is PREROLLING”.

RAW H.264 VIDEO

```
gst-launch-1.0 filesrc location=$SDL_BUILD_PATH/bin/storage/video_stream_pipe !
decodebin ! videoconvert ! xvimagesink sync=false
```

H.264 VIDEO OVER RTP

```
gst-launch-1.0 filesrc location=$SDL_BUILD_PATH/bin/storage/video_stream_pipe !
"application/x-rtp-stream" ! rtpstreamdepay ! "application/x-rtp,media=
(string)video,clock-rate=90000,encoding-name=(string)H264" ! rtph264depay !
"video/x-h264, stream-format=(string)avc, alignment=(string)au" ! avdec_h264 !
videoconvert ! ximagesink sync=false
```

RAW PCM AUDIO

```
gst-launch-1.0 filesrc location=$SDL_BUILD_PATH/bin/storage/audio_stream_pipe !
audio/x-raw,format=S16LE,rate=16000,channels=1 ! pulsesink
```

Socket Streaming

Configuration (smartDeviceLink.ini)

In the Core build folder, open `bin/smartDeviceLink.ini` and ensure the following values are set:

```
; Socket ports for video and audio streaming
VideoStreamingPort = 5050
AudioStreamingPort = 5080
...
VideoStreamConsumer = socket
AudioStreamConsumer = socket
```

GStreamer Commands

RAW H.264 VIDEO

```
gst-launch-1.0 souphttpsrc location=http://127.0.0.1:5050 ! decodebin ! videoconvert ! xvimagesink sync=false
```

H.264 VIDEO OVER RTP

```
gst-launch-1.0 souphttpsrc location=http://127.0.0.1:5050 ! "application/x-rtp-stream" ! rtpstreamdepay ! "application/x-rtp,media=(string)video,clock-rate=90000,encoding-name=(string)H264" ! rtph264depay ! "video/x-h264, stream-format=(string)avc, alignment=(string)au" ! avdec_h264 ! videoconvert ! ximagesink sync=false
```

RAW PCM AUDIO

```
gst-launch-1.0 souphttpsrc location=http://127.0.0.1:5080 ! audio/x-raw,format=S16LE,rate=16000,channels=1 ! pulsesink
```

Video Streaming States

This section describes how Core manages the streaming states of mobile applications. Only one application may stream video at a time, but audio applications may stream while in the LIMITED state with other applications.

When an app is moved to HMI level `FULL` :

- All non-streaming applications go to HMI level `BACKGROUND`
- All apps with the same App HMI Type go to `BACKGROUND`
- Streaming apps with a different App HMI Type that were in `FULL` go to `LIMITED`

When an app is moved to HMI level `LIMITED` :

- All non-streaming applications keep their HMI level
- All applications with a different App HMI Type keep their HMI level
- Applications with the same App HMI Type go to `BACKGROUND`

Additional Resources

NOTE

Livio provides an example video streaming android application.

iOS Video Streaming Guide

Android Video Streaming Guide

App Service Guidelines

This page gives a detailed look at the App Service feature in SDL Core, as well as how applications and IVI systems can integrate with the feature. For a general overview of App Services, see the App Services Overview Guide.

Terms and Abbreviations

ABBREVIATION	MEANING
ASP	App Service Provider
ASC	App Service Consumer
RPC	Remote Procedure Call

App Service RPCs

There are currently four RPCs related to app services which are available to ASCs and must be supported by every ASP. This section will describe the function of each of these RPCs, as well as the responsibilities of the ASP when they are used.

PublishAppService

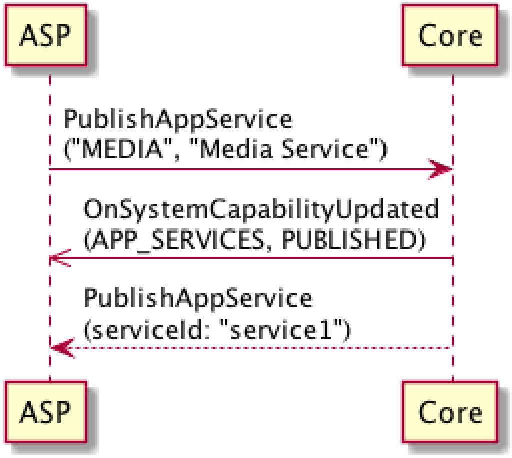
Direction: *ASP -> Core*

This request is sent by the ASP to initially create the service. This is where the service's manifest is defined, which includes the type of data provided by the service as well as what RPCs can be handled by the service.

SEQUENCE DIAGRAM

PublishAppService

View Diagram


ASP
Core
PublishAppService
("MEDIA", "Media Service")
OnSystemCapabilityUpdated
(APP_SERVICES, PUBLISHED)
PublishAppService
(serviceId: "service1")
ASP
Core

GetAppServiceData

Direction: *ASC -> Core -> ASP*

NOTE

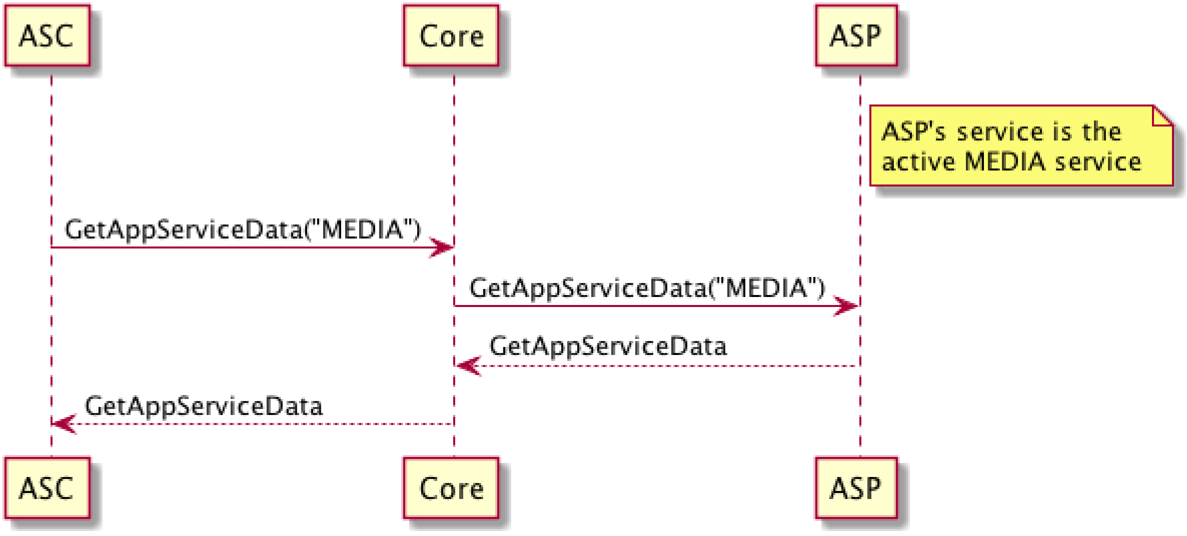
An ASP can receive this message *only* when its service is active.

The ASC can send this request to retrieve the latest app service data for a specific service type, Core will forward this request to the active service of the specified type. The ASP receiving this message is expected to respond to this message with its most recent service data.

SEQUENCE DIAGRAM

GetAppServiceData

View Diagram


ASC
Core
ASP
ASP's service is the active MEDIA service
GetAppServiceData("MEDIA")
GetAppServiceData("MEDIA")
GetAppServiceData
GetAppServiceData
ASC
Core
ASP

OnAppServiceData

Direction: *ASP -> Core -> ASC*

MUST

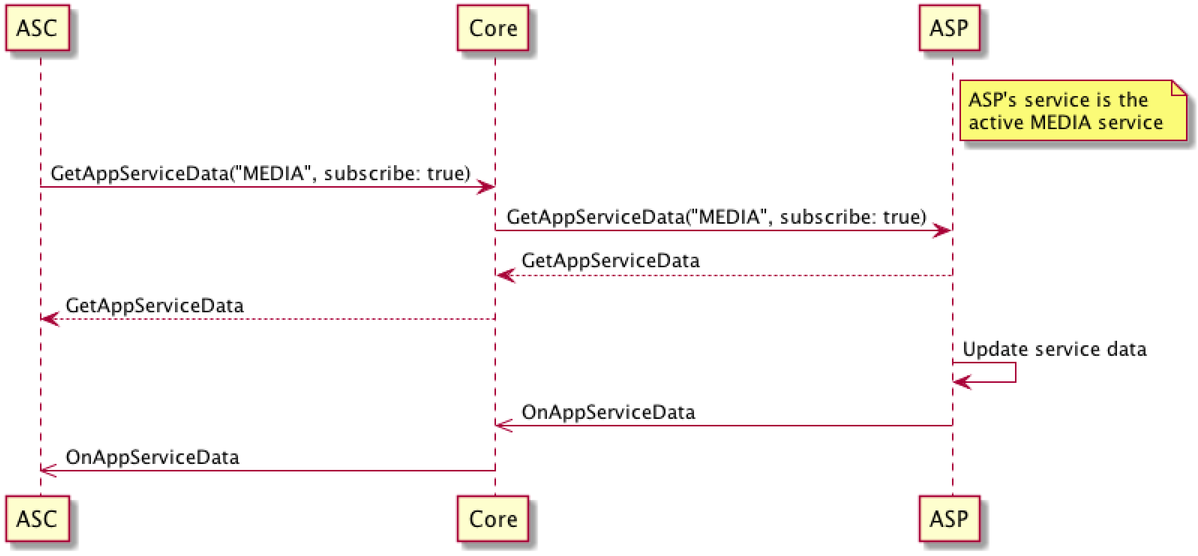
An ASP must send this message *only* when its service is active.

This notification is used to communicate updates in the app service data for a service to any ASC subscribers. The message is sent by an ASP any time that there are any significant changes to its service data while it is active *or* when its service becomes active. Core will forward this message to any ASCs that have subscribed to data for this service type.

SEQUENCE DIAGRAM

OnAppServiceData

View Diagram


ASC
Core
ASP
ASP's service is the active MEDIA service
GetAppServiceData("MEDIA", subscribe: true)
GetAppServiceData("MEDIA", subscribe: true)
GetAppServiceData
GetAppServiceData
Update service data
OnAppServiceData
OnAppServiceData
ASC
Core
ASP

PerformAppServiceInteraction

Direction: *ASC -> Core -> ASP*

NOTE

An ASP can receive this message regardless of whether its service is active, since it is directed at a specific service.

This request can be sent by an ASC to perform a service-specific function on an ASP (using the ASP's specific service ID). The API for such interactions must be defined by the ASP separately.

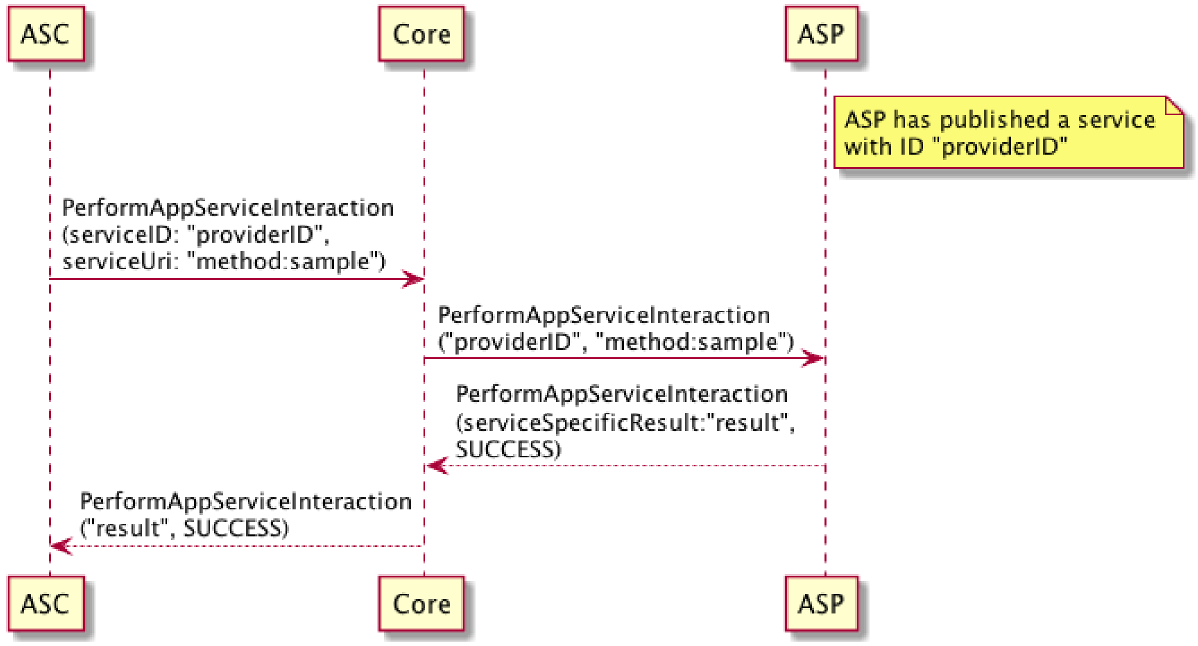
MUST

The ASP receiving this message must either process it and respond with `SUCCESS` or return an error response if the interaction was not successful.

SEQUENCE DIAGRAM

PerformAppServiceInteraction

View Diagram


ASC
Core
ASP
ASP has published a service with ID "providerID"
PerformAppServiceInteraction (serviceID: "providerID", serviceUri: "method:sample")
PerformAppServiceInteraction ("providerID", "method:sample")
PerformAppServiceInteraction (serviceSpecificResult:"result", SUCCESS)
PerformAppServiceInteraction ("result", SUCCESS)
ASC
Core
ASP

IVI App Service Integration

The App Services feature was designed to offer the same capabilities to the embedded IVI systems that are available to mobile devices. For example, the IVI's built-in radio could publish a `MEDIA` type App Service, and the embedded navigation system could publish a `NAVIGATION` type App Service.

The HMI may also act as an ASC. For example, the HMI could create a "weather widget" that subscribes to the published `WEATHER` App Service. The "weather widget" could then display weather information from the user's preferred weather service. See the App Services Overview Guide for more details on how app service data can be integrated in the IVI system by acting as an ASC.

The IVI can be configured as an ASC or ASP using a set of RPCs in the HMI API's `AppService` interface (which mirror the APIs used for mobile app services):

- `AppService.PublishAppService`
- `AppService.UnpublishAppService`
- `AppService.GetAppServiceData`
- `AppService.OnAppServiceData`
- `AppService.PerformAppServiceInteraction`

Embedded Navigation Guidelines

It is recommended that an OEM integrates App Services with their embedded navigation system to allow for a better SDL navigation experience with 3rd party applications.

If a 3rd party navigation app and the embedded navigation system are registered as navigation app services, SDL Core will be able to notify the different navigation solutions which system is activated by the user. This will prevent the possibility of two or more navigation solutions from giving the driver instructions at the same time.

MUST

A navigation ASP must stop its "in-progress" trip (if applicable) when it is notified by SDL Core that their navigation service is no longer active.

IVI-Specific RPC Messages

There are a few additional RPCs in the `AppService` interface which are needed to integrate an IVI system with the App Services feature, regardless of whether the system acts as an ASP or ASC (more information available in the HMI Integration Guidelines):

- `AppService.GetAppServiceRecords`
  - This message can be sent by the embedded IVI system to retrieve the App Service records for all published services, similar to the `GetSystemCapability (APP_SERVICES)` message available in the Mobile API. The system is expected to use this information for populating any menus within the HMI relating to App Services.
- `AppService.AppServiceActivation`
  - This message can be sent by the embedded IVI system to activate a specific service or set it as the default service for its type (usually by request of the user).
- `AppService.GetActiveServiceConsent`
  - This message is sent to the embedded IVI system whenever an ASC tries to activate an App Service (generally through `PerformAppServiceInteraction`). The system is expected to display a prompt in the HMI for the user to provide consent to activate this service, and must respond with the `activate` field populated by the user's response to this prompt.

RPC Passing

There are a number of existing RPCs which are allowed to be handled by an ASP based on service type. This feature does not apply to embedded ASPs, as messages are routed to the embedded system by default.

MEDIA

- `ButtonPress` with the following values for `buttonName`
  - `OK`
  - `PLAY_PAUSE`
  - `SEEKLEFT`
  - `SEEKRIGHT`

    - `TUNEUP`
    - `TUNEDOWN`
    - `SHUFFLE`
    - `REPEAT`

WEATHER

N/A

NAVIGATION

- `SendLocation`
- `GetWayPoints`

Flow

When RPC passing is performed with a request which relates to several components (such as ButtonPress), not all uses of this RPC will be intended for a given app service. As such, an ASP must indicate when they are unable to process a specific instance of an RPC by responding with an `UNSUPPORTED_REQUEST` response code. This informs Core that it should pass this specific request to another component or app service that handles this RPC.

This "Waterfall" flow used by Core during RPC passing is defined as follows:

1. App1 sends an RPC request to Core
2. Core checks if there is an active service which handles this RPC's function ID (ignoring any services which have already received this message)

    - If found, go to step 3
    - If not found, go to step 4

3. Core passes the raw message to the chosen ASP, waits for a response

    - If the request times out before receiving a response, return to step 2
    - If the ASP responds with result code `UNSUPPORTED_REQUEST` (indicating that it cannot handle some part of the request), return to step 2
    - If the ASP responds with a normal result code, go to step 5

4. Core handles the RPC normally, generates a response
5. Core sends the RPC response to App1

Validation

When Core passes an RPC to an ASP according to its `handledRPCs` list, it performs no additional processing on the message. This means that there is no guarantee that this message is valid according to the RPC Spec. This approach is taken specifically for forward-compatibility reasons, in case the ASP supports a newer version of the RPC Spec than Core (which could include breaking changes). As a consequence, the ASP will need to perform validation on this message itself.

Validation steps for existing passthrough RPCs:

1. Validate bounds and types of existing parameters against the RPC spec
2. Verify that mandatory parameters are present
3. For ButtonPress, verify that the `buttonName` is correctly tied to the `moduleType`

Policies

With regards to permission handling during RPC passing:

- For RPCs which are known to Core (determined by its RPC spec version), they are checked normally against the policy table. As such, the ASP can assume in this case that the app specifically has permissions to use the this RPC in its current HMI level.
- For RPCs unknown to Core, an ASC needs to be granted specific permissions by the OEM (more details here) to send this message, even if it is handled by the ASP.

Example Use Case - Sending a POI to a Navigation Provider

Before App Services were introduced, SDL applications could only send points of interest to the vehicle's embedded navigation by using the `SendLocation` RPC. The App Services feature allows an SDL app to send this same information to the active SDL navigation app instead.

Through RPC Passing, a `SendLocation` RPC request can be handled by a navigation application instead of the vehicle's navigation system. Specifically, if there is a navigation app (ASP) which can handle `SendLocation` and another SDL app (ASC) sends this message to SDL Core, it will be routed to the navigation app automatically.

ASP PREREQUISITES

1. Proper permissions must be granted to the navigation ASP in SDL Core's policy table.

    - The application acting as the ASP must have permissions to send a `PublishAppService` RPC.
    - The application's permissions must have a "NAVIGATION" object key in the "app_services" object.
    - The "NAVIGATION" object must have the functionID of `SendLocation` listed as a handled RPC.

```
{ // example sdl_preloaded_pt.json entry
    ...
    "app_policies": {
        "<provider_app_id>": {
            "keep_context": false,
            "steal_focus": false,
            "priority": "NONE",
            "default_hmi": "NONE",
            "groups": [
                "Base-4", "AppServiceProvider"
            ],
            "RequestType": [],
            "RequestSubType": [],
            "app_services": {
                "NAVIGATION": {
                    "handled_rpcs": [{"function_id": 39}]
                }
            }
        }
    }
}
```

1. The application acting as the navigation ASP must register its navigation capabilities as an app service with SDL Core via the `PublishAppService` RPC. The `AppServiceManifest` included in the request must include the function ID for `SendLocation` (39) in the `handledRPCs` array.
2. The ASP's app service must be active. This can happen a number of different ways.

    - If there is no other active navigation service, SDL Core will make an app service active when it is published.
    - If there are multiple navigation app services, SDL Core will set an app's navigation service to active whenever the app is in `HMI_LEVEL::FULL`.
    - An ASC can request to make a specific service active via the `PerformAppServiceInteraction` RPC.

ASC PREREQUISITES

Proper `SendLocation` permissions must be granted to the ASC in SDL Core's policy table.

Example sdl_preloaded_pt.json entry:

```
{
    ...
    "app_policies": {
        "<consumer_app_id>": {
            "keep_context": false,
            "steal_focus": false,
            "priority": "NONE",
            "default_hmi": "NONE",
            "groups": [
                "Base-4", "SendLocation"
            ],
            "RequestType": [],
            "RequestSubType": [],
        }
    }
}
```

USE CASE SOLUTION RPC FLOW

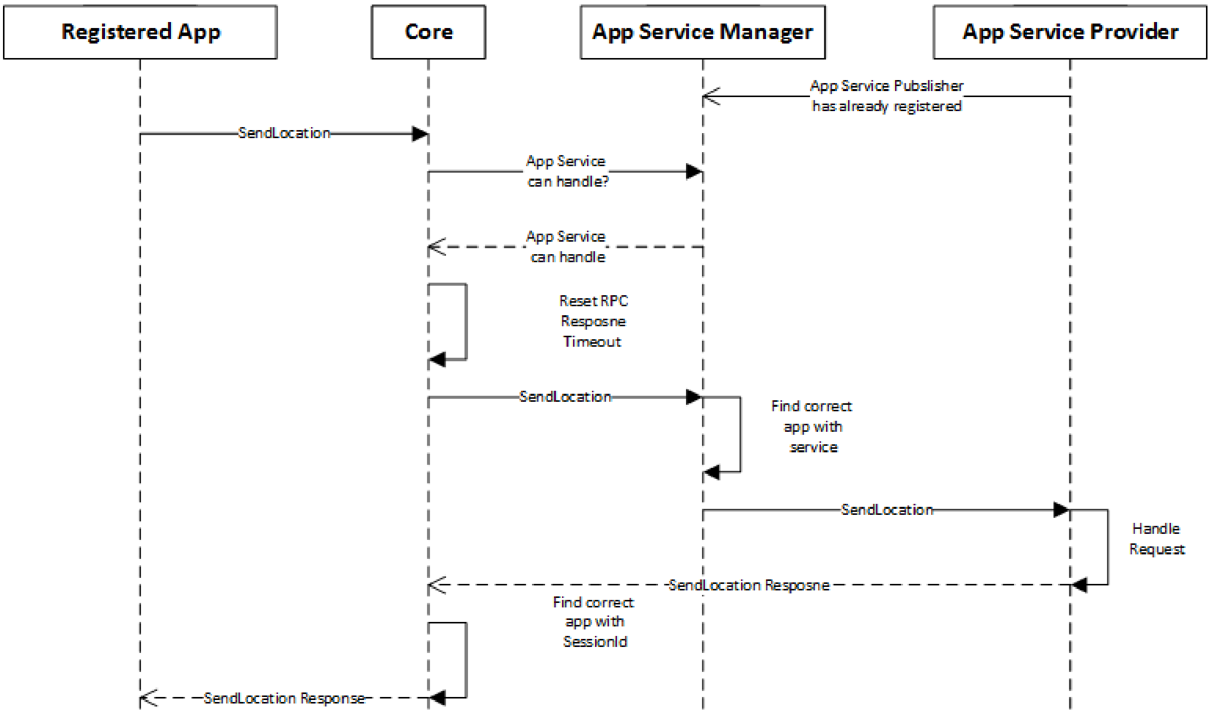
- An ASC sends a `SendLocation` RPC request to SDL Core.
- SDL Core checks if there is an active ASP that can handle the `SendLocation` RPC.
- SDL Core sends an outgoing `SendLocation` request to the active navigation ASP.

- The ASP handles the request, sets its navigation destination to the requested POI, and responds with a success to SDL Core.
- SDL Core receives the response and recognizes the message is part of an RPC Passing action. SDL Core passes the response to the navigation ASC that originated the `SendLocation` request.

SEQUENCE DIAGRAM

Example SendLocation RPC Passing

View Diagram



General Description

The Multiple Transports feature allows apps connected to SDL Core to start another connection over a different transport for certain services. For example, an app connected over Bluetooth can use WiFi as a Secondary Transport for video streaming. This guide will give an overview of the process which is used to establish a Secondary Transport connection. See SDL-0141 - Supporting Simultaneous Multiple Transports for more details on the original feature proposal.

Implementation

- After the proxy is connected to Core, it initiates another connection over a different transport.
- Core tells the proxy which transport can be used as Secondary Transport.
- The services that are allowed on the Secondary Transport are specified by Core.

> **NOTE**
>
> RPC and Hybrid services only run on the Primary Transport

There are three protocol control frames which are used in the implementation of Multiple Transports.

StartService ACK

Payload Example:

```
{
    ...
    "audioServiceTransports" : [1, 2],
    "videoServiceTransports" : [1, 2],
    "secondaryTransports" : ["TCP_WIFI"]
}
```

Core responds to the proxy's `StartService` request with additional parameters `audioServiceTransports`, `videoServiceTransports` and `secondaryTransports`.

- The `secondaryTransports` parameter contains an array of the allowed Secondary Transports for the current Primary Transport.
- `audioServiceTransports` and `videoServiceTransports` describe which services are allowed to run on which transports (Primary=1, Secondary=2, or both). The proxy uses this information and starts services only on allowed transports.
- This response is constructed by Core using the configurations defined in the SDL INI file, described in this guide.
- Since RPC and Hybrid services always run on Primary Transport, only Video and Audio services are configurable.

TransportEventUpdate

Payload Example:

```
{
    "tcpIpAddress" : "192.168.1.1",
    "tcpPort" : 12345
}
```

Core sends a `TransportEventUpdate` notification to the proxy to provide additional information required to connect over the TCP transport when it is available.

- If the `tcpIpAddress` field is empty, the Secondary Transport is unavailable and the proxy will not send a `RegisterSecondaryTransport` request.

RegisterSecondaryTransport

Using the information in the `StartService ACK` and `TransportEventUpdate` frames, the proxy sends a `RegisterSecondaryTransport` request over the Secondary Transport with the same session ID as the Primary Transport.

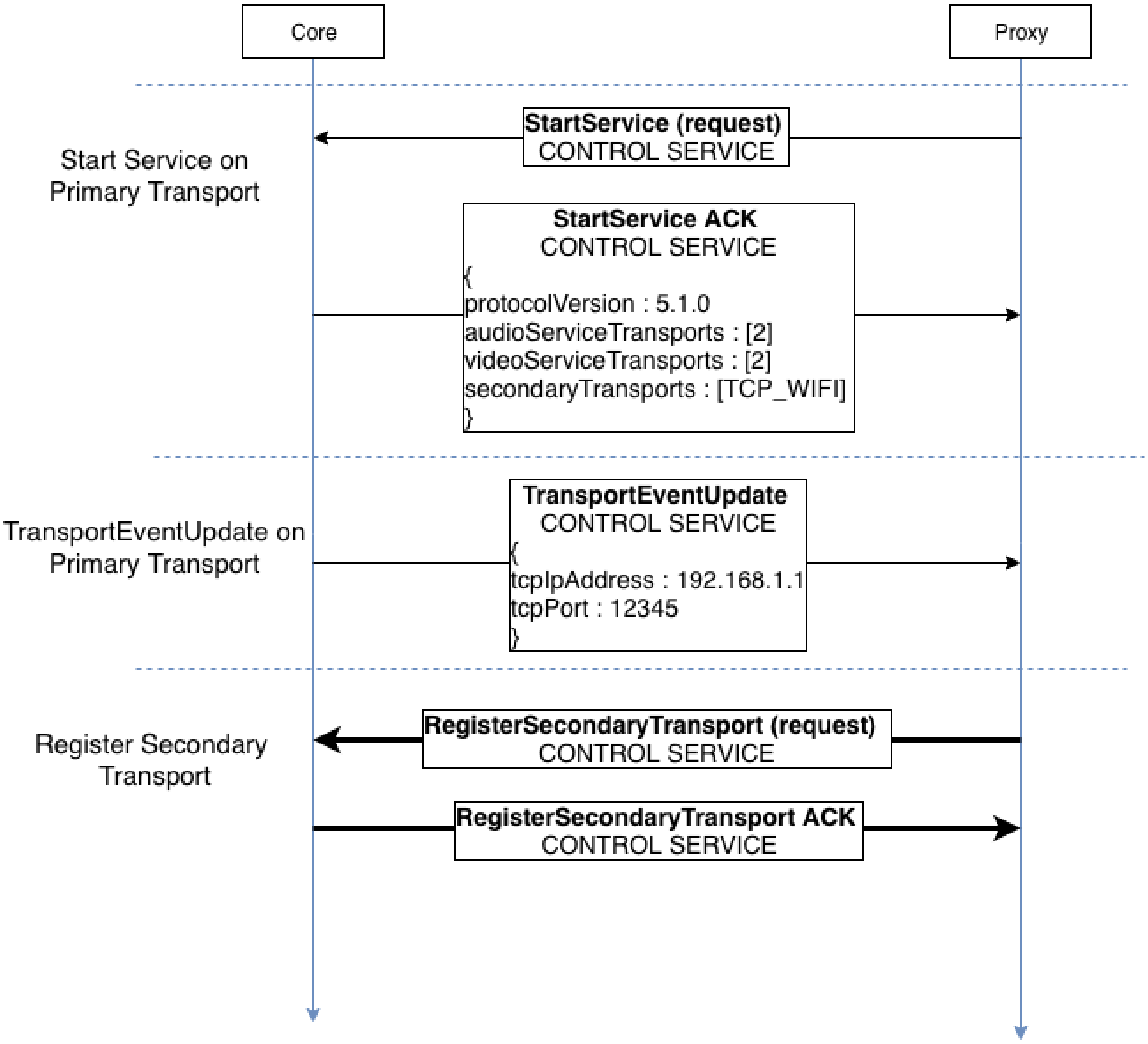
- If Core sends back a `RegisterSecondaryTransport ACK`, the proxy can start services over the Secondary Transport.

Operation Examples

SEQUENCE DIAGRAM

Start Service (WiFi as Secondary Transport)

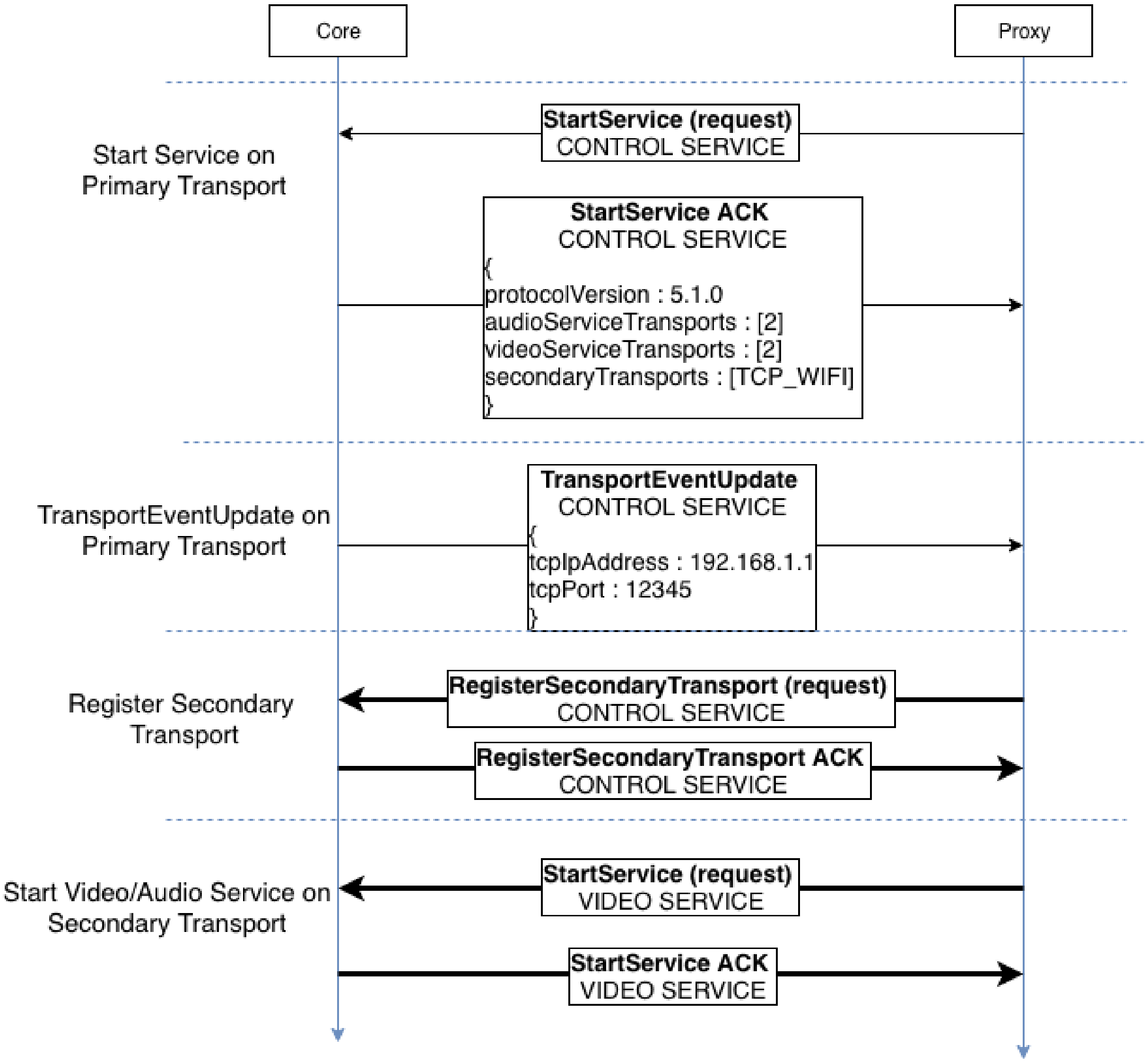
View Diagram


Core
Proxy
Start Service on Primary Transport
StartService (request)
CONTROL SERVICE
StartService ACK
CONTROL SERVICE
{
protocolVersion : 5.1.0
audioServiceTransports : [2]
videoServiceTransports : [2]
secondaryTransports : [TCP_WIFI]
}
TransportEventUpdate on Primary Transport
TransportEventUpdate
CONTROL SERVICE
{
tcpIpAddress : 192.168.1.1
tcpPort : 12345
}
Register Secondary Transport
RegisterSecondaryTransport (request)
CONTROL SERVICE
RegisterSecondaryTransport ACK
CONTROL SERVICE

SEQUENCE DIAGRAM

Start Video/Audio service (Over Secondary Transport)

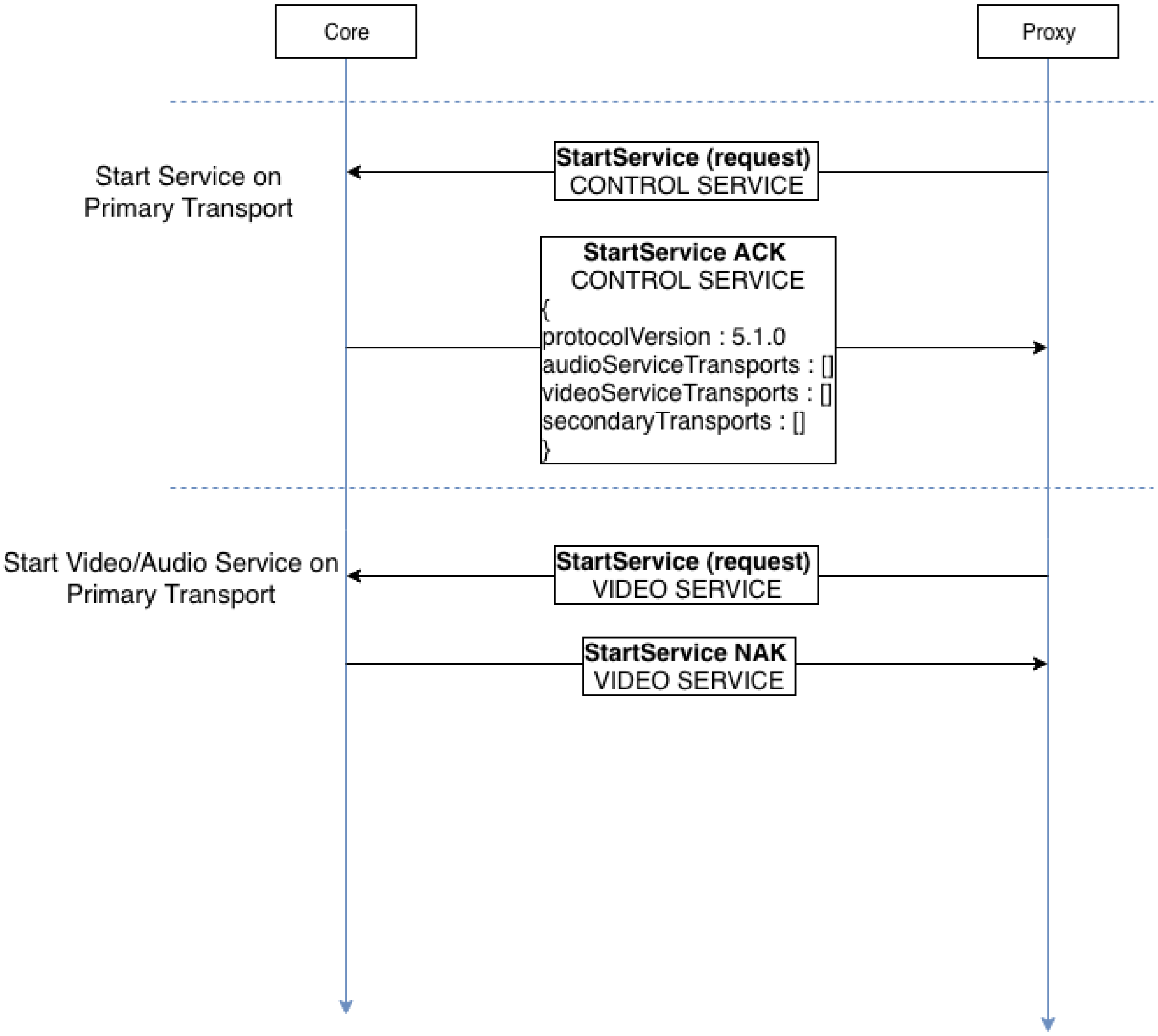
View Diagram


Core
Proxy
Start Service on Primary Transport
StartService (request)
CONTROL SERVICE
StartService ACK
CONTROL SERVICE
{
protocolVersion : 5.1.0
audioServiceTransports : [2]
videoServiceTransports : [2]
secondaryTransports : [TCP_WIFI]
}
TransportEventUpdate on Primary Transport
TransportEventUpdate
CONTROL SERVICE
{
tcpIpAddress : 192.168.1.1
tcpPort : 12345
}
Register Secondary Transport
RegisterSecondaryTransport (request)
CONTROL SERVICE
RegisterSecondaryTransport ACK
CONTROL SERVICE
Start Video/Audio Service on Secondary Transport
StartService (request)
VIDEO SERVICE
StartService ACK
VIDEO SERVICE

SEQUENCE DIAGRAM

Start Video/Audio service (No transport available)

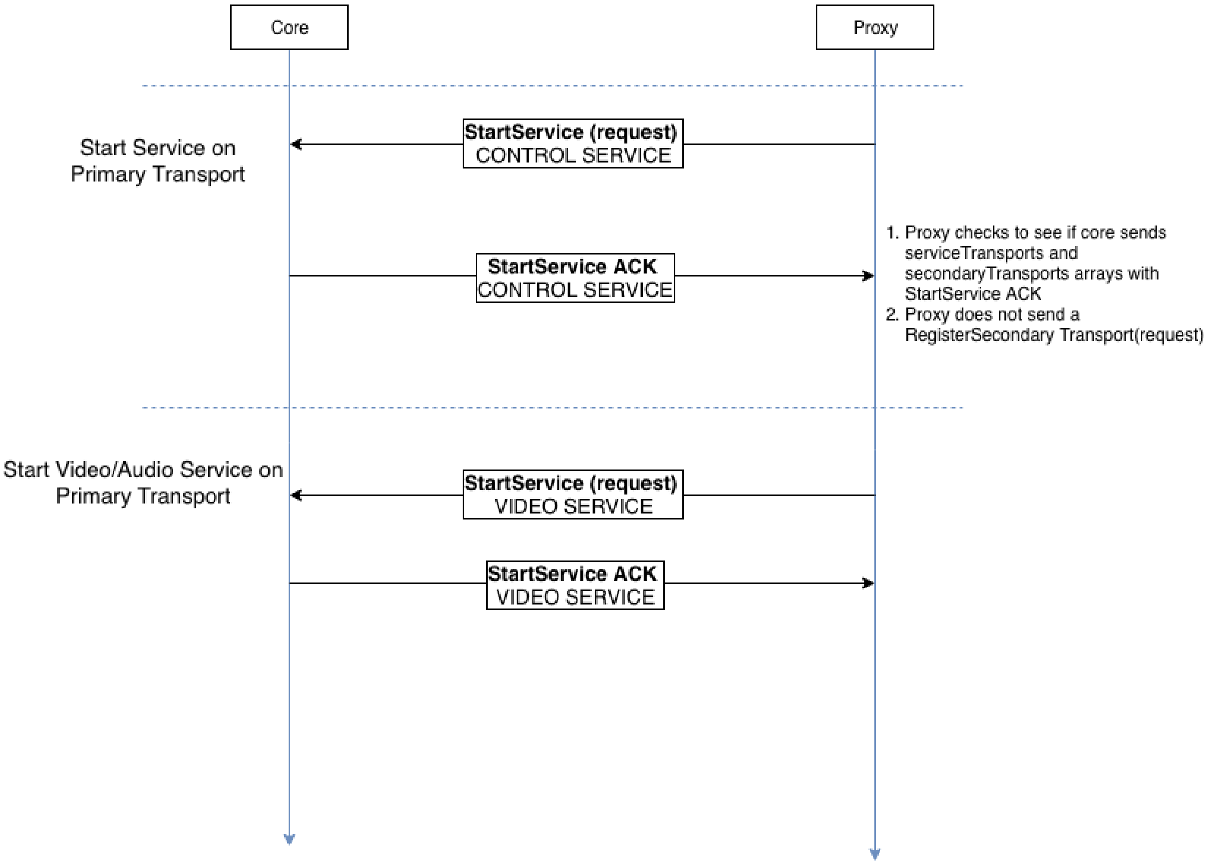
View Diagram


Core
Proxy
Start Service on Primary Transport
StartService (request) CONTROL SERVICE
StartService ACK CONTROL SERVICE
{
protocolVersion : 5.1.0
audioServiceTransports : []
videoServiceTransports : []
secondaryTransports : []
}
Start Video/Audio Service on Primary Transport
StartService (request) VIDEO SERVICE
StartService NAK VIDEO SERVICE

SEQUENCE DIAGRAM

Backwards Compatibility (New Proxy/Old Core)

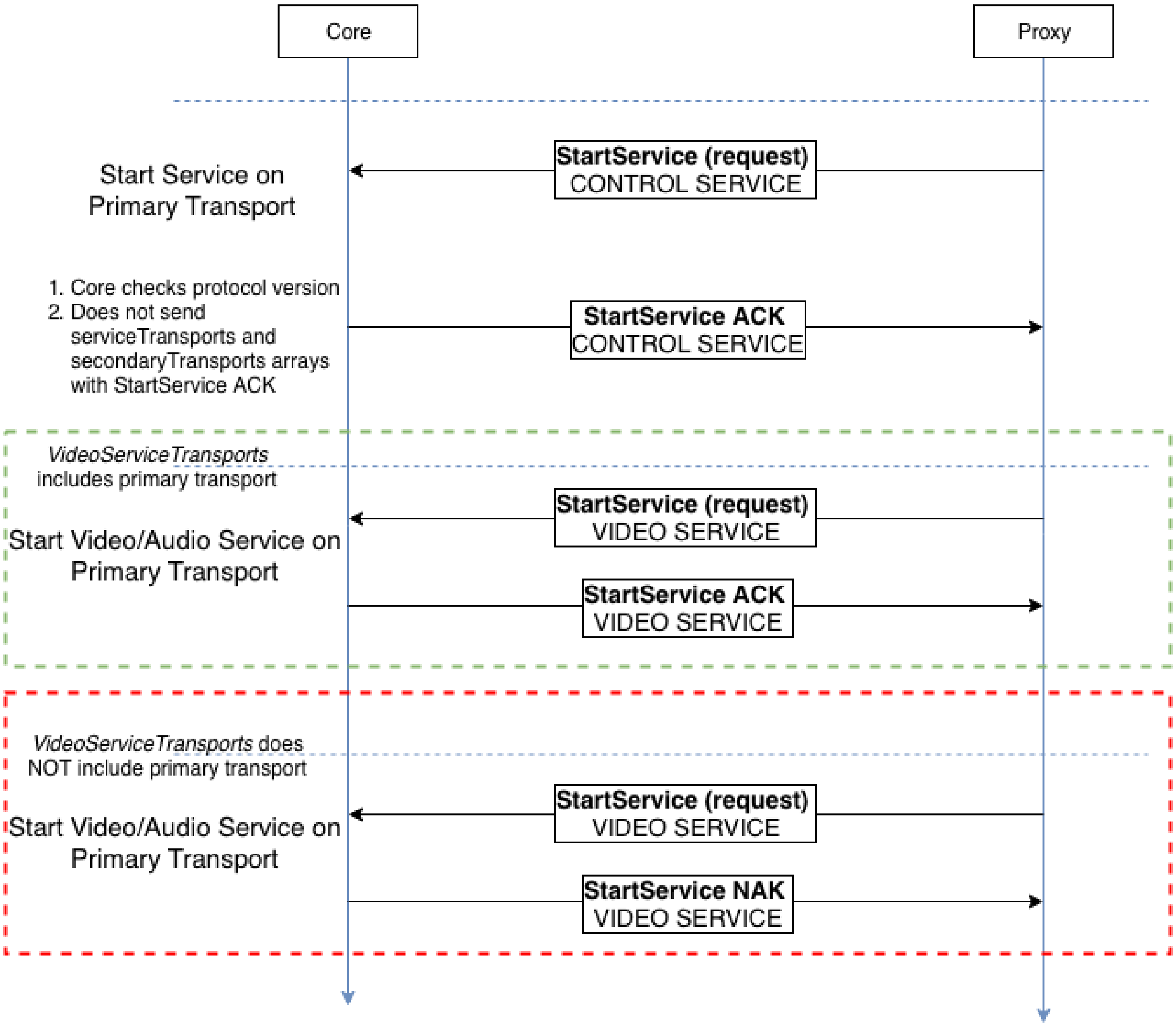
View Diagram


Core
Proxy
Start Service on Primary Transport
StartService (request) CONTROL SERVICE
StartService ACK CONTROL SERVICE
1. Proxy checks to see if core sends serviceTransports and secondaryTransports arrays with StartService ACK
2. Proxy does not send a RegisterSecondary Transport(request)
Start Video/Audio Service on Primary Transport
StartService (request) VIDEO SERVICE
StartService ACK VIDEO SERVICE

SEQUENCE DIAGRAM

Backwards Compatibility (Old Proxy/New Core)

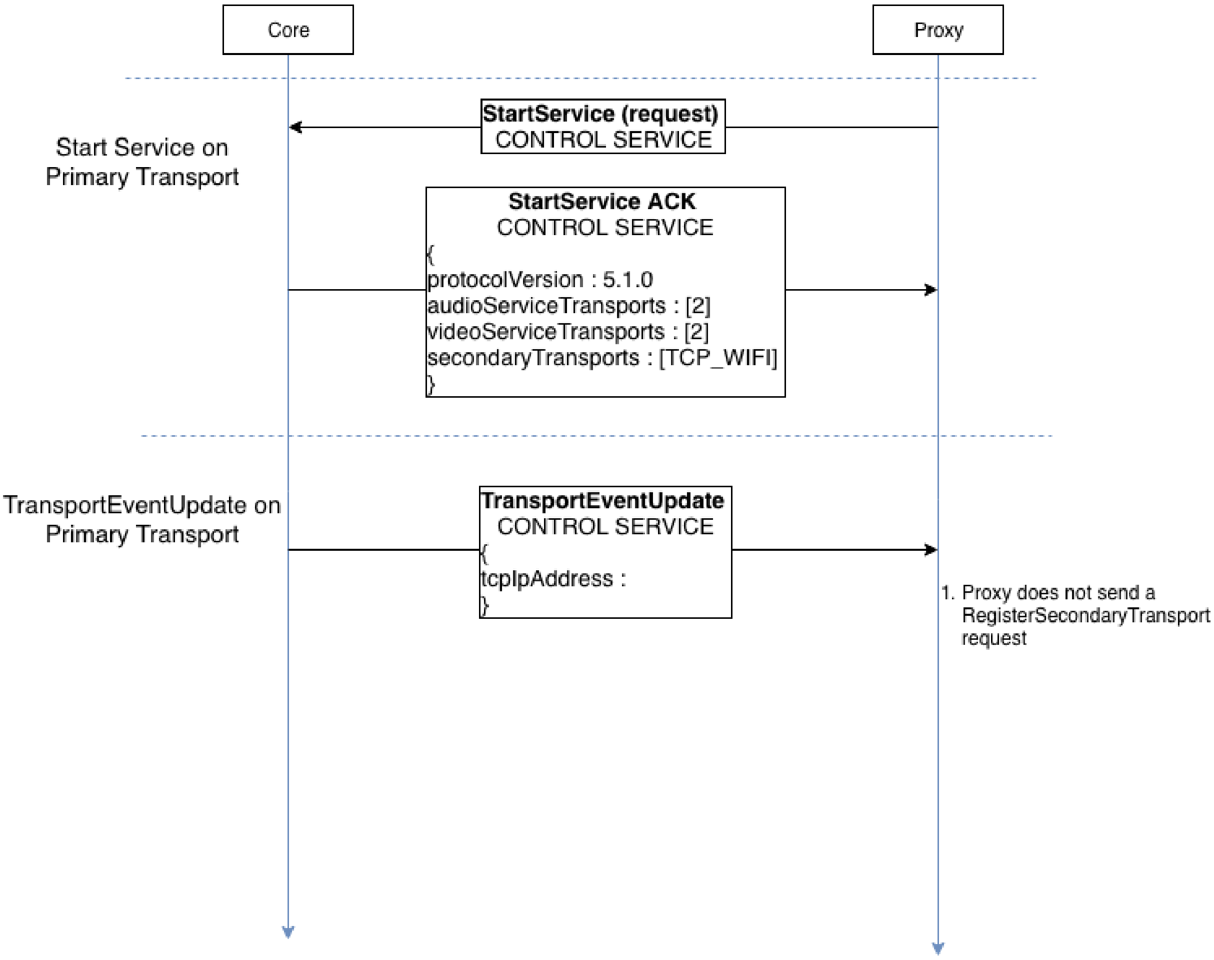
View Diagram


Core
Proxy
Start Service on Primary Transport
StartService (request) CONTROL SERVICE
1. Core checks protocol version
2. Does not send serviceTransports and secondaryTransports arrays with StartService ACK
StartService ACK CONTROL SERVICE
VideoServiceTransports includes primary transport
Start Video/Audio Service on Primary Transport
StartService (request) VIDEO SERVICE
StartService ACK VIDEO SERVICE
VideoServiceTransports does NOT include primary transport
Start Video/Audio Service on Primary Transport
StartService (request) VIDEO SERVICE
StartService NAK VIDEO SERVICE

SEQUENCE DIAGRAM

TransportEventUpdate (Secondary Transport unavailable)

View Diagram


Core
Proxy
Start Service on Primary Transport
StartService (request)
CONTROL SERVICE
StartService ACK
CONTROL SERVICE
{
protocolVersion : 5.1.0
audioServiceTransports : [2]
videoServiceTransports : [2]
secondaryTransports : [TCP_WIFI]
}
TransportEventUpdate on Primary Transport
TransportEventUpdate
CONTROL SERVICE
{
tcpIpAddress :
}
1. Proxy does not send a RegisterSecondaryTransport request

Remote Control Guide

This guide will explain how to use Remote Control within SDL. The guide will cover...

- Relevant proposals
- Relevant structs
- Relevant RPCs
- Modules and their components
- Consent rules
- Limiting permissions with policies
- Resumption

RELEVANT EVOLUTION PROPOSALS

- 0071: Remote Control Baseline
- 0099: New Remote Control Modules and Parameters
- 0105: Remote Control - Seat
- 0106: Remote Control - OnRCStatus notification
- 0160: Remote Control - Radio Parameter Update
- 0165: Remote Control - More Light Names and Status Values
- 0172: Remote Control - OnRCStatus Allowed Parameter
- 0181: Remote Control - When RC Disabled, Apps Keep HMI Level
- 0213: Remote Control - Radio and Climate Parameter Update
- 0221: Remote Control - Allow Multiple Modules per Module Type

Relevant Structs

RemoteControlCapabilities

The remote control capabilities struct contains a capabilities struct for each different remote control type.

Each capabilities struct is used to inform an app of what is available to be controlled.

- RadioControlCapabilities
- ClimateControlCapabilities
- SeatControlCapabilities
- AudioControlCapabilities
- LightControlCapabilities
- HMISettingsControlCapabilities
- ButtonCapabilities

ModuleData

The module data struct contains information used to identify a module type, and the control data associated with that module.

Each control data struct is used to observe or change the attributes of a specific module type.

- RadioControlData
- ClimateControlData
- SeatControlData
- AudioControlData
- LightControlData
- HMISettingsControlData

ModuleInfo

The module information struct is used for identifying the module and for determining who can control it.

Grid

The grid struct is used to generically describe the space within a vehicle.

Relevant RPCs

IsReady

After the `BC.IsReady` notification is received, SDL will send out an `IsReady` request for each interface. The response to this RPC just includes the boolean parameter `available` indicating if the HMI supports that interface and would like to continue to interact with it.

View **IsReady** in the HMI Documentation

GetCapabilities

Once SDL has received a positive `IsReady` response it will send a `GetCapabilities` request to the HMI. The HMI should respond with a `RemoteControlCapabilities` parameter for SDL to store and use later when a mobile application sends a `GetSystemCapability` request. This will overwrite the capabilities SDL loaded from the `hmi_capabilities.json` configuration file.

View **GetCapabilities** in the HMI Documentation

GetSystemCapability

This RPC is the starting point for an app using remote control features, it will tell you what is available to be controlled within the vehicle. GetSystemCapability is not specific to Remote Control, but a generic function used to retrieve the capabilities of multiple different modules within SDL such as navigation, video streaming or app services. However, when GetSystemCapability is called with the capability type of `REMOTE_CONTROL`, it will return the `RemoteControlCapabilities` object which in turn contains objects describing the capabilities of each remote control module present in the vehicle. These capabilities objects will contain properties like `heatedMirrorsAvailable` to indicate if a vehicle is equipped with heated mirrors, or `supportedLights` to inform SDL of which lights are available to be controlled.

View **GetSystemCapability** in the RPC Spec

GetInteriorVehicleData

GetInteriorVehicleData is used to request information about a specific module. This RPC, provided a module is specified by `moduleType` and `moduleId`, will return the status of the requested remote-control module. This RPC can also be used to subscribe to updates of a module's status via the `subscribe` parameter. If this non-mandatory parameter is set to true, the head unit will register `OnInteriorVehicleData` notifications for the requested module. Conversely, if this parameter is set to false, the head unit will unregister `OnInteriorVehicleData` notifications for the requested module.

NOTE

If an application sends GetInteriorVehicleData (subscribe=true, moduleType=MODULE1), but the application is already subscribed on MODULE1 module type, SDL will respond with a `WARNINGS` resultCode because of the double subscription.

View **GetInteriorVehicleData** in the RPC Spec or the HMI Documentation

OnInteriorVehicleData

OnInteriorVehicleData is a notification sent out by the HMI when an update is made to a remote control module. An app can subscribe to these notifications via GetInteriorVehicleData. This RPC will come with a `ModuleData` structure identifying the changed module and containing the control data object with the new state.

View **OnInteriorVehicleData** in the RPC Spec or the HMI Documentation

SetInteriorVehicleData

SetInteriorVehicleData is used to set the values of a remote control module by passing in a `ModuleData` structure. The `moduleType` and `moduleId` fields are used to identify the targeted module, and the changes in the respective control data object are applied to that module.

View **SetInteriorVehicleData** in the RPC Spec or the HMI Documentation

OnRemoteControlSettings

OnRemoteControlSettings is used to notify SDL when passengers of a vehicle change the remote control settings via the HMI. This includes allowing or disallowing Remote Control or changing the access mode that will be used for resource allocation.

View **OnRemoteControlSettings** in the HMI Documentation

OnRCStatus

OnRCStatus is a notification sent out by SDL when an update is made to a remote control module's availability. When SDL either allocates a module to an app, or deallocates it from an app, SDL will send OnRCStatus to both the application and the HMI. This notification contains two lists, one describing the modules that are allocated to the application and the other describing the free modules that can be accessed by the application. This notification also contains an `allowed` parameter, which indicates to apps whether or not Remote Control is currently allowed. If `allowed` is false, both module lists will be empty.

View **OnRCStatus** in the RPC Spec or the HMI Documentation

GetInteriorVehicleDataConsent

GetInteriorVehicleDataConsent is a request used to reserve remote control modules. If a module does not allow multiple access, only the application that requested consent first will be able to interact with that module. Otherwise, if the module does allow multiple access, the rules specified in the Consent section) apply. This request requires a `moduleType` and an array of `moduleId`s to identify the target modules. Core will reply with an array of booleans indicating the consent for each requested `moduleId` where true signals allowed and vice versa.

View **GetInteriorVehicleDataConsent** in the RPC Spec or the HMI Documentation

ReleaseInteriorVehicleDataModule

ReleaseInteriorVehicleDataModule is a request used to free a remote control module once an application is finished interacting with it. This request requires a `moduleType` and `moduleId` to identify the target module.

View **ReleaseInteriorVehicleDataModule** in the RPC Spec

SetGlobalProperties

SetGlobalProperties is a request sent by a mobile app to inform SDL of a user's location within the vehicle. The request includes a `userLocation` parameter which contains a grid. The location of a user is important for SDL to know so it can determine whether or not a user is within a module's service area.

View **SetGlobalProperties** in the RPC Spec or the HMI Documentation

Remote Control Modules

Climate

The climate module consists of climate sub-modules represented by a `ClimateControlCapabilities` object. Each sub-module exposes many aspects of a car's climate controls, such as setting the desired temperature or turning on the heated windshield.

Radio

The radio module consists of radio sub-modules represented by a `RadioControlCapabilities` object. Each sub-module exposes many aspects of a car's radio controls, such as setting the desired frequency and band the radio is operating on.

Seat

The seat module consists of seat sub-modules represented by a `SeatControlCapabilities` object. Each sub-module exposes many aspects of a car's seat controls, such as setting the back tilt angle and the massage mode.

Audio

The audio module consists of audio sub-modules represented by a `AudioControlCapabilities` object. Each sub-module exposes many aspects of a car's audio controls, such as setting the volume or modifying the equalizer settings.

Light

The light module does not contain any sub-modules but instead has an array of `LightCapabilities` objects, each identified by a `LightName`. This module exposes the ability to modify attributes such as the brightness and color of each light.

HMI Settings

The HMI settings module does not contain any sub-modules and is represented by an `HMISettingsControlCapabilities` object. This module exposes the ability to set the desired temperature and distance units as well as toggle the display mode of the HMI between night and day.

Button

Button is an interesting remote control component because it is not a remote control module. `RemoteControlCapabilities` includes an array of `ButtonCapabilities` structs which describe either a physical button or a softbutton. A mobile app may send a `Button`

`Press` RPC with the `ButtonName` and `moduleId` from any of these `ButtonCapabilities` to perform an action on another remote control module.

Consent

The behavior of module allocation in SDL Core is shown in the following table:

NOTE

The driver is always considered to be within the service area.
SDL will assume actions performed by the driver are consented to by the driver.
Resources can only be acquired by apps in HMI level full.

USER LOCATION	ALLOW MULTIPLE ACCESS	REQUESTED MODULE STATE	ACCESS MODE	SDL ACTION
out of service area	any	any	any	disallow
in service area	any	free	any	allow
in service area	false	in use	any	disallow
in service area	true	in use	auto allow	allow
in service area	true	in use	auto deny	disallow
in service area	true	in use	ask driver	ask driver

REQUESTED MODULE STATE

- "free" indicates no application currently holds the requested resource
- "in use" indicates that an application currently holds the requested resource
- "busy" indicates at least one RC RPC request is currently executing and has yet to finish

Policies

You can take a look at the Remote Control section of the policies guide to see how remote control permissions are defined.

Resumption

Interior Vehicle Data Subscriptions

During the data resumption process, SDL sends `GetInteriorVehicleData(subscribe=true)` requests to the HMI and stores data received from the HMI in a cache.

If during resumption the HMI responds with error to a `GetInteriorVehicleData` request or responds with SUCCESS to a `GetInteriorVehicleData` but with parameter `isSubscribed=false`, SDL reverts already subscribed data and fails resumption for related application(s), removing information about this subscription.

For more information about how SDL handles resumption, you can take a look at the Application Data Resumption guide.

RPC Encryption

RELEVANT EVOLUTION PROPOSALS

- 0207: RPC Message Protection

Introduction

This guide will cover the basic setup required to enable and utilize RPC Encryption within SDL Core. For more information about the feature, please take a look at the RPC Encryption Overview Guide.

Encryption Setup

Generate Self Signed Certificate

- Create private key:

```
openssl genrsa -out client.key 2048
```

- Create CSR:

```
openssl req -new -key client.key -out client.req -subj
'/C=US/ST=MI/L=Detroit/O=SDL/OU=HeadUnit/CN=client/emailAddress=sample@sdl
```

- Create Public Certificate:

```
openssl x509 -hash -req -in client.req -signkey client.key -out client.cert -days 10000
```

Configure SDL Core

INI FILE MODIFICATIONS

- Copy client.key and client.cert into your SDL Core `build/bin` directory. Delete any existing key, cert/crt, or pem files.

In your `build/bin` directory run:

```
c_rehash .
```

- Set the certificate and key file path for SDL in smartDeviceLink.ini. The INI Configuration has more information about the properties in the INI file.

```
; Certificate and key path to pem file
CertificatePath = client.cert
KeyPath        = client.key
```

- If you are using self signed certificates set VerifyPeer to false.

```
; Verify Mobile app certificate (could be used in both SSLMode Server and Client)
VerifyPeer  = false
```

POLICY TABLE MODIFICATIONS

The policy table can be modified to enforce encryption on certain RPCs. These modifications can be made in your sdl_preloaded_pt.json before launching Core or by updating the policy table while Core is running via a PTU

- Add "encryption_required": true to a functional group in the functional_groupings section

```
...
  "functional_groupings": {
    ...
    "EncryptedRPCs": {
      "encryption_required" : true,
      "rpcs":{
        "AddCommand": {
          "hmi_levels": ["BACKGROUND",
          "FULL",
          "LIMITED"]
        },
        "Alert": {
          "hmi_levels": ["BACKGROUND",
            "FULL",
            "LIMITED"]
        },
        ...
      }
    },
    ...
  }
...
```

- Add `"encryption_required": true` to an application in the `app_policies` section

```
...
  "app_policies": {
    ...
    "appId": {
      "keep_context": false,
      "steal_focus": false,
      "priority": "NONE",
      "default_hmi": "NONE",
      "groups": ["Base-4", "EncryptedRPCs"],
      "RequestType": [],
      "RequestSubType": [],
      "encryption_required": true
    },
    ...
  }
...
```

JSON EXAMPLE

Below is a possible policy table configuration requiring an app to use encryption for a specific functional group.

```
"functional_groupings": {
    "EncryptedAddCommand": {
        "encryption_required" : true,
        "rpcs":{
            "AddCommand": {
                "hmi_levels": ["BACKGROUND",
                "FULL",
                "LIMITED"]
            }
        }
    },
    ...
},
...
"app_policies": {
    "<PUT_APP_ID_HERE>": {
        "keep_context": false,
        "steal_focus": false,
        "priority": "NONE",
        "default_hmi": "NONE",
        "groups": ["Base-4", "EncryptedAddCommand"],
        "RequestType": [],
        "RequestSubType": [],
        "encryption_required": true
    },
    ...
}
```

Additional Resources

- Android Encryption Guide
- iOS Encryption Guide

Service Status Update

General Description

This guide will explain how the `BasicCommunication.OnServiceUpdate` RPC is used within SDL Core. At a high level, this RPC is used by SDL Core to inform the HMI of the status of the system or what steps to take in case of an error. For example, when a mobile navigation application is activated and sends a request to start a Video Service, a series of steps are taken: getting the system time, performing a policy table update, and finally decrypting and validating certificates. SDL Core sends `BC.OnServiceUpdate` notifications to the HMI throughout each of these steps to provide information on the status of the system. These notifications may cause the HMI to display a popup providing this status information in a readable format to the user, or inform the user of what steps to take in case of an error.

Parameters

The `OnServiceUpdate` notification has three parameters:

serviceType

This parameter is mandatory and will contain a value from the `ServiceType` enum, indicating the type of service that this update is for:

- VIDEO
- AUDIO
- RPC

serviceEvent

This parameter is not mandatory and will be a value from the `ServiceEvent` enum, indicating the status of the StartService request:

- REQUEST_RECEIVED
- REQUEST_ACCEPTED
- REQUEST_REJECTED

reason

This parameter is not mandatory and will be a member of the `ServiceStatusUpdateReason` enum, indicating the type of error that occurred while attempting to start the service:

- PTU_FAILED
  - the system was unable to get a required Policy Table Update
- INVALID_CERT
  - the security certificate was invalid or expired
- INVALID_TIME
  - the system was unable to get a valid `SystemTime` from the HMI
- PROTECTION_ENFORCED
  - the system configuration (ini file) requires a service to be protected, but the app attempted to start an unprotected service
- PROTECTION_DISABLED
  - the system started an unprotected service when the app requested a protected service

appID

This parameter is not mandatory but will be included with each request after the `RegisterAppInterface` message for this application has been received.

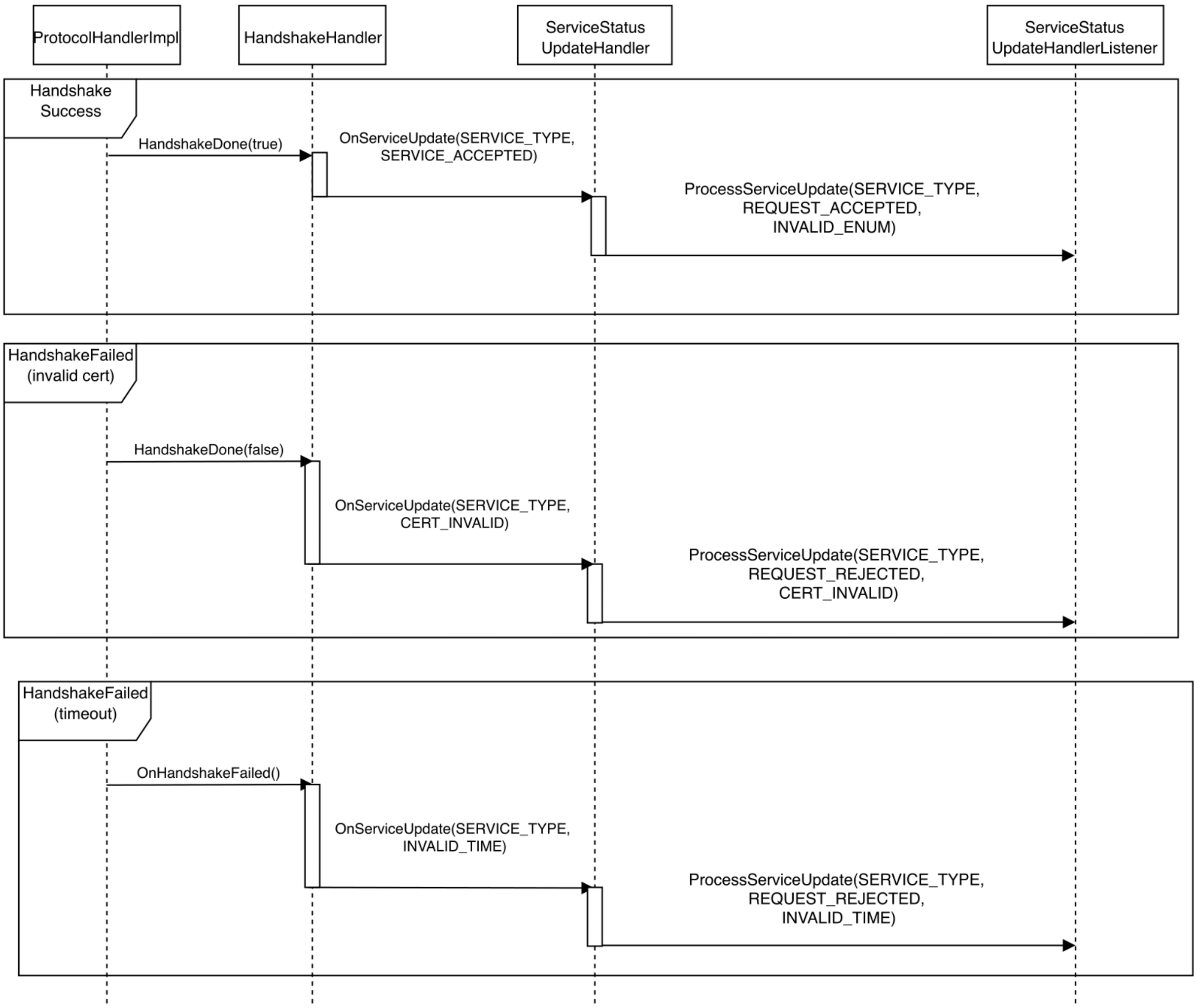
Flow Diagrams

More documentation on the message flow for `BC.OnServiceUpdate` and its parameters can be found in the HMI Integration Guidelines.
To better understand how the `OnServiceUpdate` notification is propagated within SDL Core, please take a look at the following Sequence Diagrams:

SEQUENCE DIAGRAM

OnServiceUpdate Handshake Flow

View Diagram

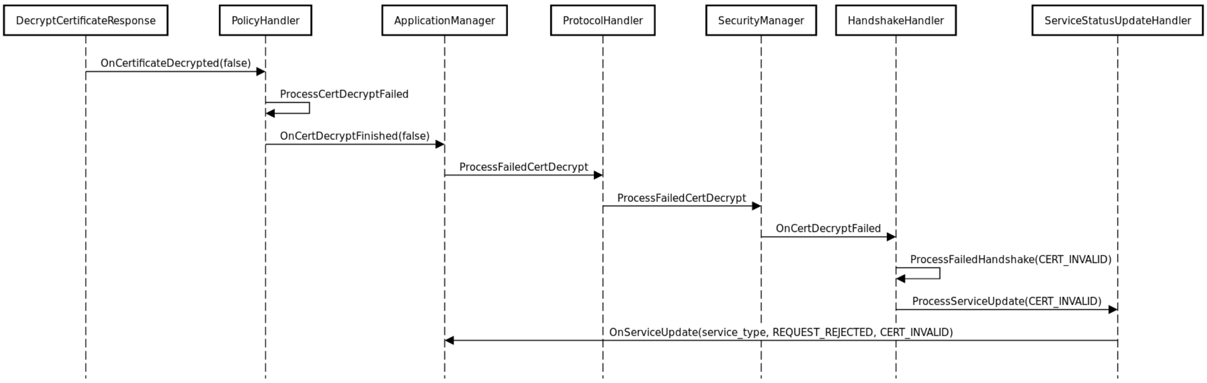


ProtocolHandlerImpl
HandshakeHandler
ServiceStatus UpdateHandler
ServiceStatus UpdateHandlerListener
Handshake Success
HandshakeDone(true)
OnServiceUpdate(SERVICE_TYPE, SERVICE_ACCEPTED)
ProcessServiceUpdate(SERVICE_TYPE, REQUEST_ACCEPTED, INVALID_ENUM)
HandshakeFailed (invalid cert)
HandshakeDone(false)
OnServiceUpdate(SERVICE_TYPE, CERT_INVALID)
ProcessServiceUpdate(SERVICE_TYPE, REQUEST_REJECTED, CERT_INVALID)
HandshakeFailed (timeout)
OnHandshakeFailed()
OnServiceUpdate(SERVICE_TYPE, INVALID_TIME)
ProcessServiceUpdate(SERVICE_TYPE, REQUEST_REJECTED, INVALID_TIME)

SEQUENCE DIAGRAM

OnServiceUpdate Invalid Certificate

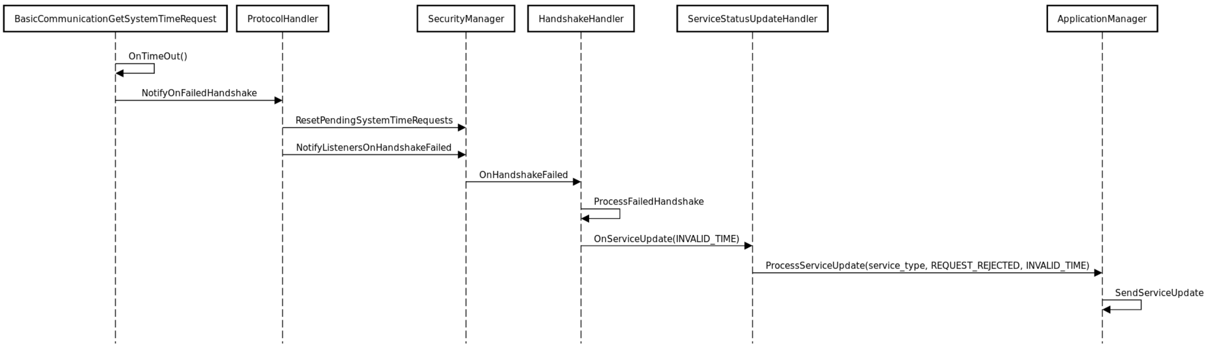
View Diagram


DecryptCertificateResponse
PolicyHandler
ApplicationManager
ProtocolHandler
SecurityManager
HandshakeHandler
ServiceStatusUpdateHandler
OnCertificateDecrypted(false)
ProcessCertDecryptFailed
OnCertDecryptFinished(false)
ProcessFailedCertDecrypt
ProcessFailedCertDecrypt
OnCertDecryptFailed
ProcessFailedHandshake(CERT_INVALID)
ProcessServiceUpdate(CERT_INVALID)
OnServiceUpdate(service_type, REQUEST_REJECTED, CERT_INVALID)

SEQUENCE DIAGRAM

OnServiceUpdate GetSystemTime Failed

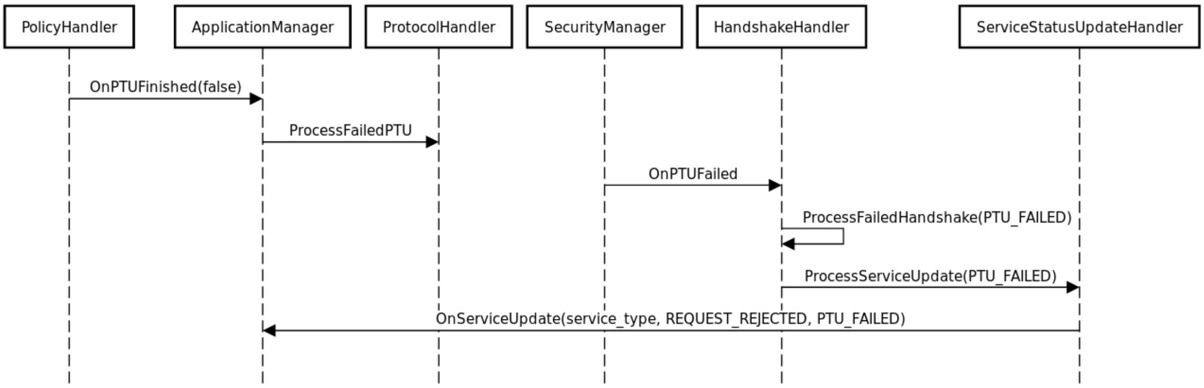
View Diagram


BasicCommunicationGetSystemTimeRequest
ProtocolHandler
SecurityManager
HandshakeHandler
ServiceStatusUpdateHandler
ApplicationManager
OnTimeOut()
NotifyOnFailedHandshake
ResetPendingSystemTimeRequests
NotifyListenersOnHandshakeFailed
OnHandshakeFailed
ProcessFailedHandshake
OnServiceUpdate(INVALID_TIME)
ProcessServiceUpdate(service_type, REQUEST_REJECTED, INVALID_TIME)
SendServiceUpdate

SEQUENCE DIAGRAM

OnServiceUpdate Policy Table Update Failed

View Diagram


PolicyHandler
ApplicationManager
ProtocolHandler
SecurityManager
HandshakeHandler
ServiceStatusUpdateHandler
OnPTUFinished(false)
ProcessFailedPTU
OnPTUFailed
ProcessFailedHandshake(PTU_FAILED)
ProcessServiceUpdate(PTU_FAILED)
OnServiceUpdate(service_type, REQUEST_REJECTED, PTU_FAILED)

Smart Objects

Smart Objects are a recursive custom dynamic data structure within SDL Core which can be used to easily store and manipulate complex data. Developers can use Smart Objects to create containers for most primitive types, as well as arrays and maps.

Usage

The current implementation of Smart Objects contains definitions that allow it to store the following data types: bool, int, long, double, char, string (both as char* and std::string), array, and map.

The `ns_smart_device_link::ns_smart_objects::CSmartObject` class also defines a set of methods that can be used to represent the stored object value as a desired type.

Example Usage:

```
ns_smart_device_link::ns_smart_objects::SmartObject obj;

obj[0] = 1;

obj[1] = true;

obj[2] = 'a';

obj[3] = 3.14;

int i = obj[0].asInt();

bool b = obj[1].asBool();

char c = obj[2].asChar();

double d = obj[3].asDouble();
```

Validation

Smart Objects also include a validation/normalization mechanism called a Schema, which is similarly structured to the Mobile/HMI API XML schemas. This object allows the client to validate any existing Smart Object data structure. The process of validation includes both type and value validation for the Smart Object.

To validate a Smart Object, a Schema needs to be applied to it. To "apply" a Schema means that the Schema will modify the object to "normalize" its data. Applying the Schema can be done by using the `ns_smart_device_link::ns_smart_objects::CSmartSchema::applySchema` method. Internally, the apply method for a Schema triggers the apply method of every Schema Item within the object, and currently only modifies enum Schema Items. When this method is called on a enum Schema Item, it will try to convert the string representation of the object to one the item's predefined enum values.

The validation of a specific Smart Object can be triggered by using the `ns_smart_device_link::ns_smart_objects::CSmartSchema::validate` method. Internally, the validate method triggers the respective validate method for each Schema Item in the object in order to perform validation.

To "unapply" modifications done by the apply step, the Schema's `ns_smart_device_link::ns_smart_objects::CSmartSchema::unapplySchema` method can be used. This reverts all enum values back to their string representations.

Schema Structure

Every Schema is constructed using objects called Schema Items. Each Schema Item defines the type of a specific data structure as well as any restrictions for that structure's values.

In order to create a new Schema (a new object of class `ns_smart_device_link::ns_smart_objects::CSmartSchema`), you must first must define all of the required Schema Items for this object. These Schema Items can have a recursive tree structure, and each node and leaf of that tree defines structural rules for some part of the Smart Object data structure.

Schema Items are represented as class hierarchy. The base class for all Schema Items is the `ns_smart_device_link::ns_smart_objects::ISchemaItem` class. This base class defines a generic validation interface for all Schema Items.

- To define special elements which always fail or succeed the validation step, there are two special Schema Items: `ns_smart_device_link::ns_smart_objects::CAlwaysTrueSchemaItem` or `ns_smart_device_link::ns_smart_objects::CAlwaysFalseSchemaItem`.
- `ns_smart_device_link::ns_smart_objects::CBoolSchemaItem` is used for boolean values and has no parameters, meaning that it only verifies that the Smart Object contains an actual boolean value.
- `ns_smart_device_link::ns_smart_objects::TNumberSchemaItem` is a template Schema Item that can be used for both integer and floating point values. In addition to simple type verification, it is possible to set an optional min and max value range for this item.
- `ns_smart_device_link::ns_smart_objects::TEnumSchemaItem` is used to verify any custom client-defined enum. It is constructed using a list of these custom enum values.
- `ns_smart_device_link::ns_smart_objects::CStringSchemaItem` is used to verify a string value. In addition to simple type verification, it is possible to set an optional min and max string length for this item.
- `ns_smart_device_link::ns_smart_objects::CArraySchemaItem` provides validation for an array containing values with another Schema Item. It can be used to verify an array with optional size bounds.

- `ns_smart_device_link::ns_smart_objects::CObjectSchemaItem` is used to verify a map structure. Each Schema Item of this type includes a list of child Schema Items with associated keys. All other Schema Item types make up the leaf nodes of the validation tree for this Schema Item.

After the creation of all required Schema Items, it is then possible to create a Schema. A Schema can be initialized not only by raw root Schema Item, but also by special abstraction called a Member (defined by the `ns_smart_device_link::ns_smart_objects::SMember` class). So every root item (`ns_smart_device_link::ns_smart_objects::CObjectSchemaItem`) firstly should be wrapped as Member. This wrapping process is also used to set the "mandatory" property for each Member. After each Member has been constructed, the root Schema Item is then used to construct the final Schema.

Currently all Schemas are generated by the InterfaceGenerator tool. The Schema for an SDL mobile message has following structure:

```
message
 |
 -- params
 |      |
 |      -- function_id
 |      |
 |      -- message_type
 |      |
 |      -- correlation_id
 |      |
 |      -- protocol_version
 |      |
 |      -- protocol_type
 |
 -- msg_params
        |
        -- (function-specific leaf item)
       ...
```

Schema Construction Example

```
namespace messageType {
/**
 * @brief Enumeration messageType.
 *
 *
 *            Enumeration linking message types with function types in WiPro protocol.
 *            Assumes enumeration starts at value 0.
 *
 */
enum eType {
  /**
   * @brief INVALID_ENUM.
   */
  INVALID_ENUM = -1,

  /**
   * @brief request.
   */
  request = 0,

  /**
   * @brief response.
   */
  response = 1,

  /**
   * @brief notification.
   */
  notification = 2
};
} // messageType
...
namespace FunctionID {
/**
 * @brief Enumeration FunctionID.
 *
 * Enumeration linking function names with function IDs in SmartDeviceLink
protocol. Assumes enumeration starts at value 0.
 */
enum eType {
  /**
   * @brief INVALID_ENUM.
   */
  INVALID_ENUM = -1,

  /**
   * @brief RESERVED.
   */
  RESERVED = 0,

  /**
   * @brief RegisterAppInterfaceID.
   */
  RegisterAppInterfaceID = 1,
  ...
};
} // FunctionID
...
// Struct member success.
//
//  true if successful; false, if failed
std::shared_ptr<ISchemaItem> success_SchemaItem =
CBoolSchemaItem::create(TSchemaItemParameter<bool>());

// Struct member resultCode.
//
// See Result
std::shared_ptr<ISchemaItem> resultCode_SchemaItem =
TEnumSchemaItem<Result::eType>::create(resultCode_allowed_enum_subset_values
 TSchemaItemParameter<Result::eType>());

// Struct member info.
//
// Provides additional human readable info regarding the result.
std::shared_ptr<ISchemaItem> info_SchemaItem =
CStringSchemaItem::create(TSchemaItemParameter<size_t>(1),
TSchemaItemParameter<size_t>(1000), TSchemaItemParameter<std::string>());

Members schema_members;
schema_members["success"] = SMember(success_SchemaItem, true, "1.0.0", "",
false, false);
schema_members["resultCode"] = SMember(resultCode_SchemaItem, true, "1.0.0", "",
false, false);
schema_members["info"] = SMember(info_SchemaItem, false, "1.0.0", "", false, false);

Members params_members;
params_members[ns_smart_device_link::ns_json_handler::strings::S_FUNCTION_ID]
= SMember(TEnumSchemaItem<FunctionID::eType>::create(function_id_items), true);
params_members[ns_smart_device_link::ns_json_handler::strings::S_MESSAGE_TYPE
 =
SMember(TEnumSchemaItem<messageType::eType>::create(message_type_items),
true);
params_members[ns_smart_device_link::ns_json_handler::strings::S_PROTOCOL_VEF
 = SMember(TNumberSchemaItem<int>::create(), true);
params_members[ns_smart_device_link::ns_json_handler::strings::S_PROTOCOL_TYF
 = SMember(TNumberSchemaItem<int>::create(), true);
params_members[ns_smart_device_link::ns_json_handler::strings::S_CORRELATION_I
 = SMember(TNumberSchemaItem<int>::create(), true);

Members root_members_map;
root_members_map[ns_smart_device_link::ns_json_handler::strings::S_MSG_PARAMS
 = SMember(CObjectSchemaItem::create(schema_members), true);
root_members_map[ns_smart_device_link::ns_json_handler::strings::S_PARAMS] =
SMember(CObjectSchemaItem::create(params_members), true);
return CSmartSchema(CObjectSchemaItem::create(root_members_map));
```

WebEngine Application Guide

- What is a WebEngine app?
- What is the new transport that WebEngine apps use?

WebEngine Apps

A WebEngine app is a web application that runs within the vehicle. This is made possible by an OEM hosted "app store" which distributes approved "app bundles." The HMI will decompress these app bundles and launch the entrypoint which will use the SDL JavaScript (JS) library to interact with SDL Core.

App bundles are zip compressed archives containing the following files:

MANIFEST.JS

manifest.js is a javascript file that exports the following application properties:

- `entrypoint`
  - A relative path within the bundle to the HTML file that will be launched by the HMI
  - This HTML file must include the manifest.js file as a script
- `appIcon`
  - A relative path to the app icon within the app bundle
- `appId`

    - The `policyAppId` of this application
- `appName`
    - The app name that should be displayed in the app store or in the app list
- `category`
    - The primary `appHMIType` of the WebEngine application
- `additionalCategories`
    - Additional `appHMIType`s of the WebEngine application
- `locales`
    - A map of other languages to alternate names and icons
- `appVersion`
    - The current version of the application
- `minRpcVersion`
    - The minimum supported RPC spec version
- `minProtocolVersion`
    - The minimum supported protocol spec version

SDL.JS

sdl.js contains the SDL JS library used to interact with SDL Core.

OTHER SUPPORTING JAVASCRIPT FILES

All other HTML / JS files used to run the application.

Example Application

To see an example WebEngine application, take a look at the example in the javascript suite on GitHub.

Launching a WebEngine App

In order for a WebEngine application to appear on the SDL app list, the HMI must notify SDL Core about newly installed WebEngine applications by sending a `BasicCommunication.SetAppProperties`. The properties sent in this request will be stored in Core's policy table and the information will stay persistent between ignition cycles.

> **NOTE**
>
> Not all parameters in the `AppProperties` struct apply to WebEngine Apps. WebEngine Apps should omit `endpoint` and `authToken`.

When the user activates a WebEngine application, the HMI will use information from the manifest.js to launch the entrypoint HTML (our development HMIs do this by creating an invisible iframe). Here, the app will begin execution on the head unit and eventually call `RegisterAppInterface`. When the HMI receives an `OnAppRegistered` notification signalling that the WebEngine app has successfully registered, the HMI should then send Core an `ActivateApp` request.

WebSocket Server Transport

In order to support the WebEngine feature, a WebSocket server transport was added to SDL Core. This contrasts to the WebSocket client transport in SDL Core that is used by Java Cloud applications. When the HMI launches a WebEngine application, it will provide Core's hostname and port as query parameters to the entrypoint of the WebEngine application. This transport supports both secure and non-secure WebSocket communication, which is also determined by a query parameter passed to the entrypoint HTML file of the WebEngine application.

These are the accepted values for the sdl-transport-role parameter:

- ws-server
- ws-client
- wss-server
- wss-client
- tcp-server
- tcp-client

Example URL with query parameters: `file://somewhere/HelloSDL/index.html?sdl-host=localhost&sdl-port=12345&sdl-transport-role=wss-server`

SECURED WEBSOCKET CONNECTIONS

WebSocket server transport will only run if either all three of these are valid or if none are provided:

- WSServerCertificatePath (path to WebSocket server certificate)
- WSServerKeyPath (path to WebSocket server private key path)
- WSServerCACertificatePath (path to CA certificate)

If all three are provided, SDL Core will use WebSocket Secure, otherwise, Core will use regular WebSocket communication. These values can be set in the smartDeviceLink.ini configuration file.

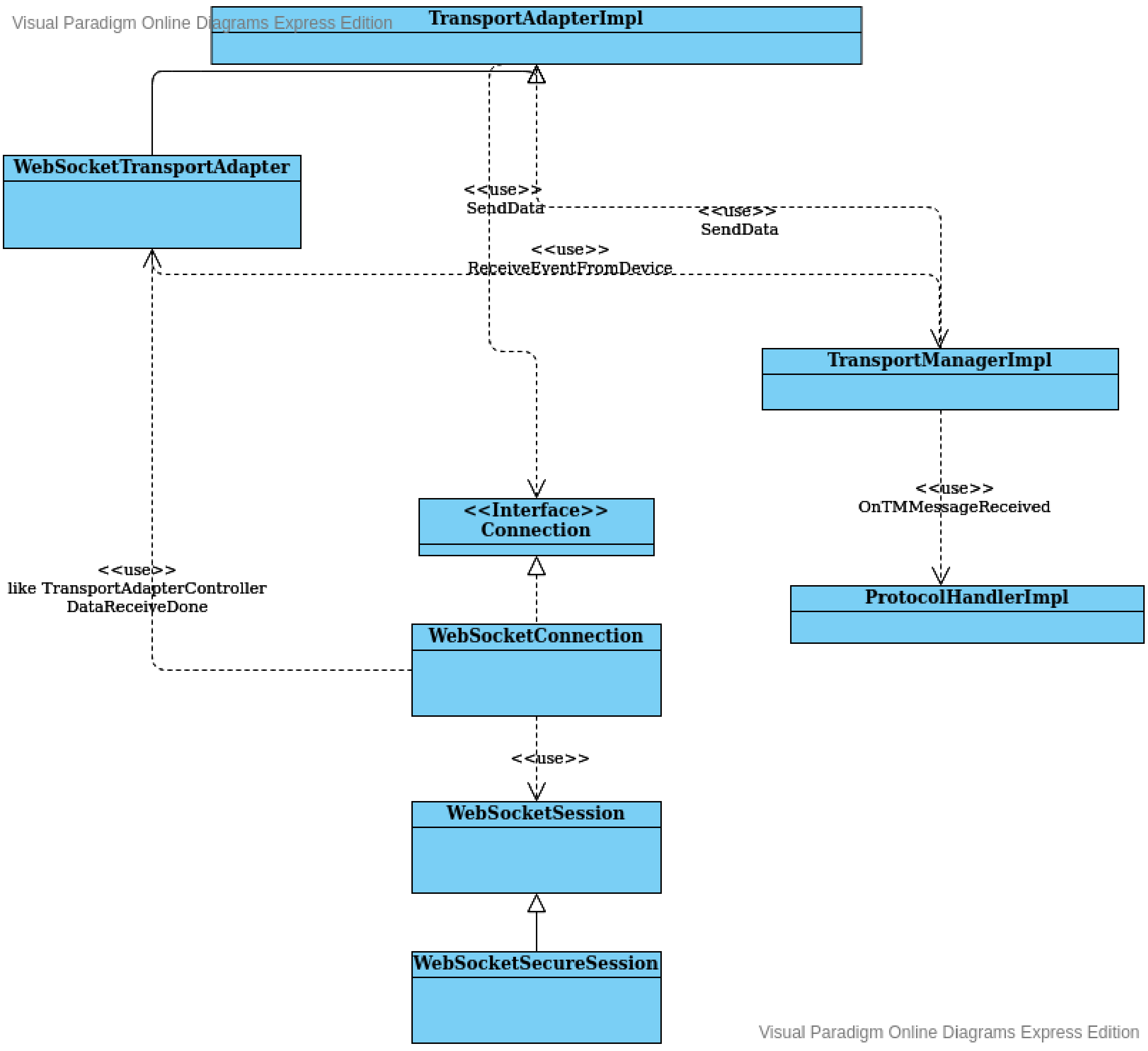
WEBSOCKET SERVER COMPONENT HIERARCHY

Please refer to the following diagram which describes the hierarchy of transport components for the WebSocket Server transport adapter.

SEQUENCE DIAGRAM

WebSocket Server Hierarchy

View Diagram



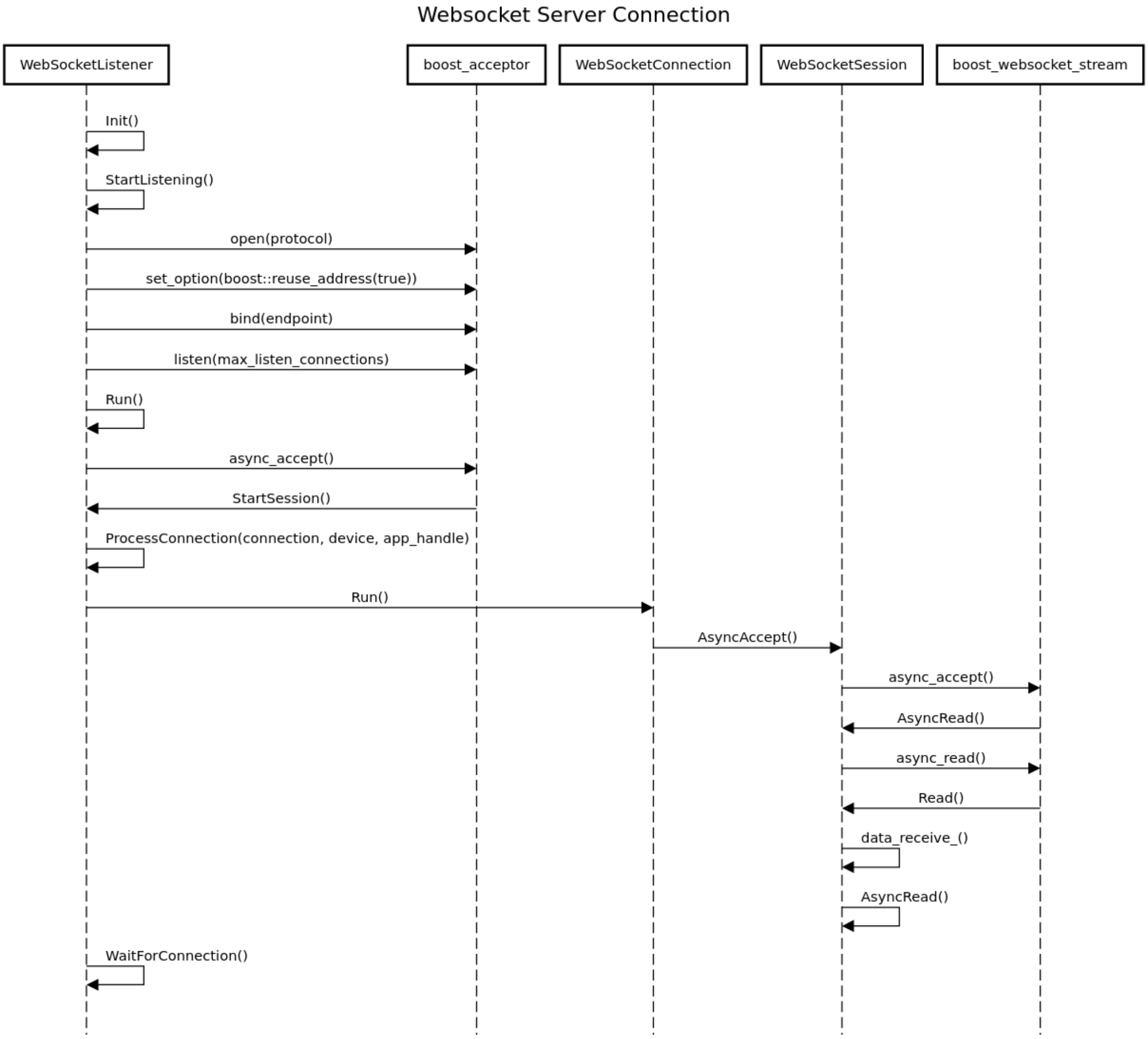
WEBSOCKET SERVER INIT SEQUENCE

Please refer to the following diagram that describes the initialization sequence when SDL Core is started.

SEQUENCE DIAGRAM

WebSocket Server Connection Sequence

View Diagram



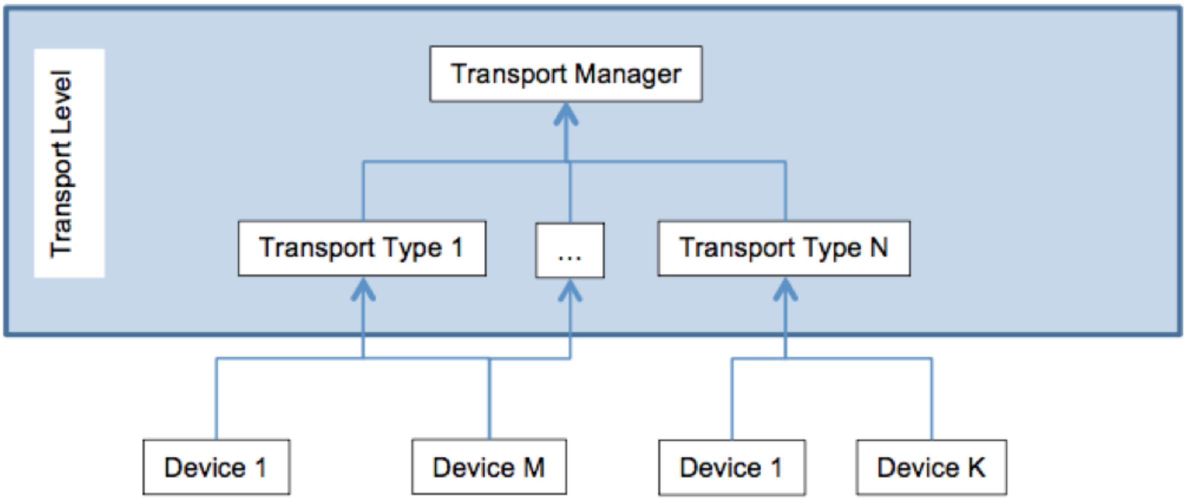
Transport Programming Guide

This guide will explain how transports work in SDL Core. We will highlight the responsibilities of each interface as well as look at some examples of transports already implemented in SDL Core. First let's take a look at Figure 1, a diagram showing the hierarchy of the main transport components and then we will work our way down the diagram describing each component.

SEQUENCE DIAGRAM

Figure 1: Transport Overview

View Diagram



Transport Manager

The Transport Manager is responsible for routing commands and messages between the transport adapters and other major components in SDL Core. A Transport Manager can contain any number of Transport Adapters, each of which is responsible for handling communication via one type of transport, such as TCP or Bluetooth. The Transport Manager also contains data necessary to handle its responsibilities, such as a mapping of each device to the Transport Adapter it uses to communicate. Other components within SDL Core are also able to register a Transport Manager Listener with the manager, which will receive events from the Transport Manager. The default Transport Manager follows the singleton pattern, but this is not necessary if you would like to use a custom solution.

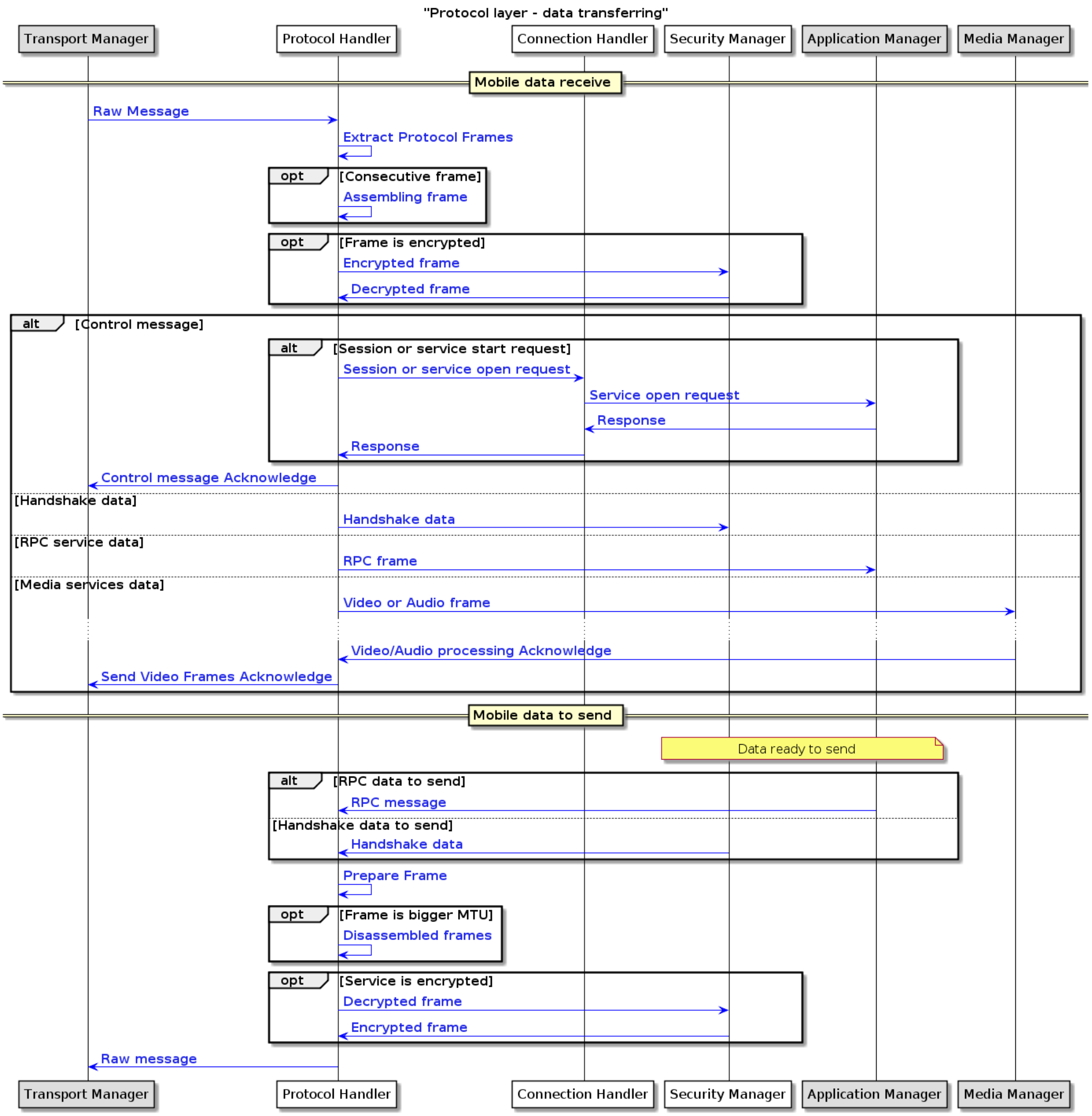
TRANSPORT MANAGER RESPONSIBILITIES

This diagram describing how data is transferred within the Protocol Layer can help you understand the responsibilities of the Transport Manager.

SEQUENCE DIAGRAM

Figure 2: Protocol Layer Data Transfer

View Diagram

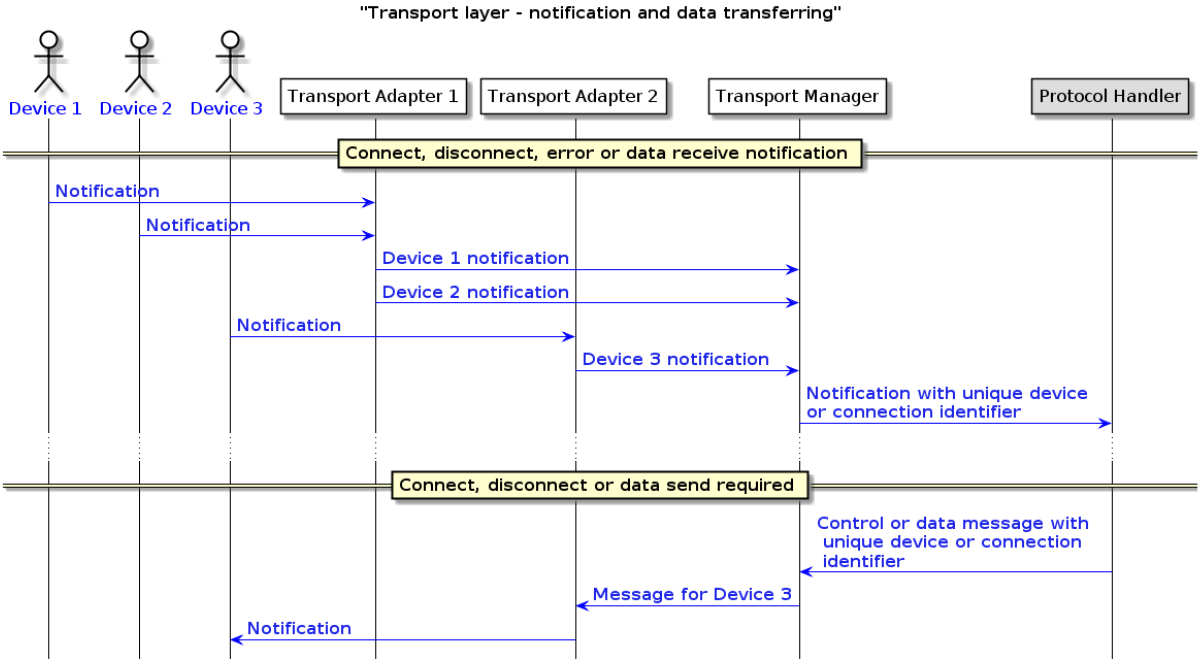


This diagram describing how data is transferred within the Transport Layer can help you understand the responsibilities of the Transport Manager.

SEQUENCE DIAGRAM

Figure 3: Transport Layer Data Transfer

View Diagram

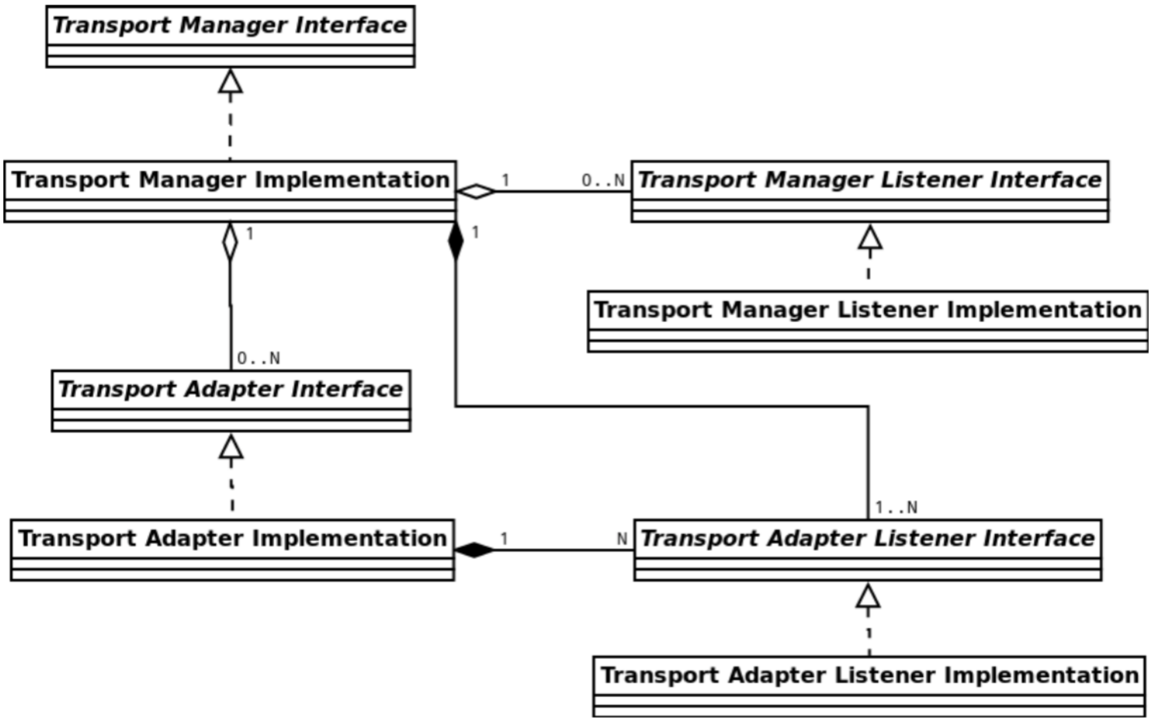

"Transport layer - notification and data transferring"
Device 1
Device 2
Device 3
Transport Adapter 1
Transport Adapter 2
Transport Manager
Protocol Handler
Connect, disconnect, error or data receive notification
Notification
Notification
Device 1 notification
Device 2 notification
Notification
Device 3 notification
Notification with unique device or connection identifier
Connect, disconnect or data send required
Control or data message with unique device or connection identifier
Message for Device 3
Notification

TRANSPORT MANAGER INHERITANCE STRUCTURE

SEQUENCE DIAGRAM

Figure 4: Transport Manager UML Diagram

View Diagram


Transport Manager Interface
Transport Manager Implementation
1
0..N
Transport Manager Listener Interface
1
1
Transport Manager Listener Implementation
0..N
Transport Adapter Interface
1..N
Transport Adapter Implementation
1
N
Transport Adapter Listener Interface
Transport Adapter Listener Implementation

NOTE

Classes named like *Impl only *represent* implementations of the abstract sub classes and may not be named the same in the SDL Core project.

UML Refresher

- Aggregation: Solid line with open diamond
- Composition: Solid line with filled diamond
- Inheritance: Dotted line with open arrow
- Dependency: Dotted line with two prong arrow

Transport Adapter

Each Transport Adapter is responsible for one specific type of connection, such as TCP or Bluetooth. Similar to Transport Managers, other components in Core are able to register a Transport Adapter Listener with a Transport Adapter to later receive events from the

Adapter such as `OnConnectDone`. The Transport Adapter will contain the code to connect and disconnect devices, as well as send and receive data. Depending on the transport type, a transport adapter may implement sub-components, called workers, such as a Device Scanner, a Client Connection Listener, or a Server Connection Factory. Currently, Transport Adapters are registered with the Transport Manager within the TransportManagerDefault::Init method; you can add code here to include your custom Transport Adapter. Depending on your implementation, most of the functionality of the Transport Adapter will likely live in the workers. Two big functions that will for sure need to be implemented in a Transport Adapter are Store() and Restore() which are used to save and resume the state of the Adapter when there is an unexpected disconnect or SDL Core is restarted. In the case of the TCP Transport Adapter, the Store() function will save a list of devices' names and addresses, along with the applications each device was running and their corresponding port number. When resuming, Restore() will reconnect to the devices saved in the last state and resumes communication with the applications on each device.

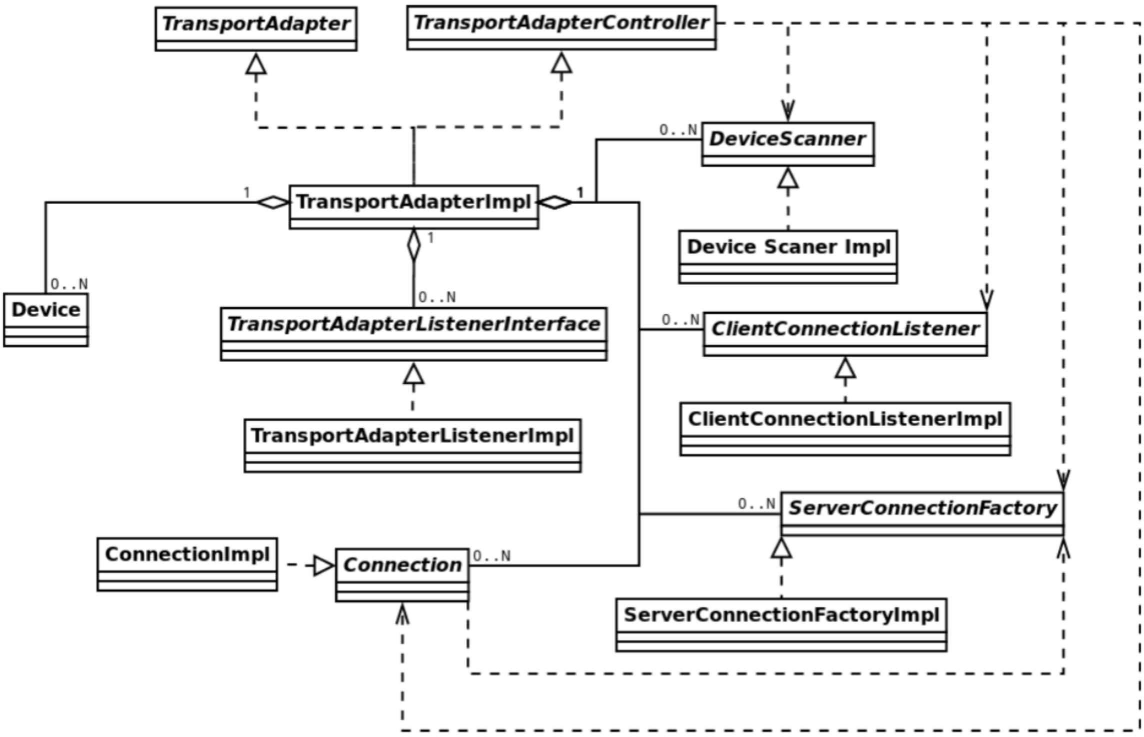
```
// tcp_transport_adapter.cc
// Code has been heavily simplified and whitespace has been added for readability.

void TcpTransportAdapter::Store() const {
  Json::Value devices_dictionary;

  for (Device* tcp_device : GetDeviceList()) {
    if (!tcp_device) { continue; } // device could have been disconnected

    Json::Value device_dictionary;
    device_dictionary["name"] = tcp_device->name();
    device_dictionary["address"] = tcp_device->in_addr();

    Json::Value applications_dictionary;
    for (ApplicationHandle app_handle : tcp_device->GetApplicationList()) {
      if (FindEstablishedConnection(tcp_device->unique_device_id(),
                                    app_handle)) {
        int port = tcp_device->GetApplicationPort(app_handle);
        if (port != -1) {  // don't want to store incoming applications
          applications_dictionary.append(std::string(port));
        }
      }
    }

    if (!applications_dictionary.empty()) {
      device_dictionary["applications"] = applications_dictionary;
      devices_dictionary.append(device_dictionary);
    }
  }

  Json::Value& dict = last_state().get_dictionary();
  dict["TransportManager"]["TcpAdapter"]["devices"] = devices_dictionary;
}
```

TRANSPORT ADAPTER INHERITANCE STRUCTURE

SEQUENCE DIAGRAM

Figure 5: Transport Adapter UML Diagram

View Diagram



NOTE

Classes named like *Impl only *represent* implementations of the abstract sub classes and may not be named the same in the SDL Core project.

UML Refresher

- Aggregation: Solid line with open diamond
- Composition: Solid line with filled diamond
- Inheritance: Dotted line with open arrow
- Dependency: Dotted line with two prong arrow

Transport Adapter Workers

The Client Connection Listener implements receiving a connection that is originated by a device. This will typically wait for connection from a device, then establish that connection, finally alerting the Transport Manager via the Transport Manager Listener of the newly connected device and app IDs. The TCP transport adapter has a good example implementation of a Client Connection Listener.

The Server Connection Factory implements a connection that is originated from Core. For example, Core reaches out to a predefined web address to start a cloud websocket application. This type of communication requires that the Transport Adapter knows of the device and application in advance. When this connection is created, the Transport Adapter will alert the Transport Manager of the new devices and applications in a similar fashion to other workers. USB and Bluetooth are additional examples that implement this sub-component.

The Device Scanner is responsible for scanning for new devices to connect with. When a device is found, this worker is responsible for alerting the Transport Adapter, as well as alerting the Transport Manager via the Transport Adapter Listener. Next, the Transport Manager will instruct the Transport Adapter to connect with the devices. The Bluetooth Transport Adapter has a great example implementation of the Device Scanner that search for bluetooth services advertising the SDL bluetooth UUID.

Depending on what your type of transport is, whether Core will be the server or the client, you will likely implement either the Client Connection Listener or the Server Connection Factory.

CLIENT CONNECTION LISTENER

Using the TCP Transport Adapter as an example for a client connection listener implementation, let's take a look at `Init()`.

```
// tcp_client_listener.cc
// Code has been simplified and whitespace has been added for readability.
// Thank you to Sho Amano for the helpful comments.

TransportAdapter::Error TcpClientListener::Init() {
  thread_stop_requested_ = false;

  if (!IsListeningOnSpecificInterface()) {
    // Network interface is not specified. We will listen on all interfaces
    // using INADDR_ANY. If socket creation fails, we will treat it an error.
    socket_ = CreateIPv4ServerSocket(port_);
    if (-1 == socket_) {
      LOG("Failed to create TCP socket");
      return TransportAdapter::FAIL;
    }
  } else {
    // Network interface is specified and we will listen only on the interface.
    // In this case, the server socket will be created once
    // NetworkInterfaceListener notifies the interface's IP address.
    LOG("TCP server socket will listen on " << designated_interface_
                  << " once it has an IPv4 address.");
  }

  if (!interface_listener_->Init()) {
    if (socket_ >= 0) {
      close(socket_);
      socket_ = -1;
    }
    return TransportAdapter::FAIL;
  }

  initialized_ = true;
  return TransportAdapter::OK;
}
```

The Init() function is called one time to prepare the Transport Adapter for its work. If the initialization work within this function succeeds, `initialized_` should be set to true. In the case of the TCP Client Listener, the Init() method will create the socket and initialize the interface listener. It is worth noting that the interface listener is specific to the TCP Client Listener and contains the lower level code to accept TCP connections such as the socket() and bind() syscalls. A similar component is not necessary. The opposite of Init() is the Terminate() method which handles shutting down the transport adapter. On the TCP Client Listener, this involves destroying the socket and de-initializing the interface listener.

```
// tcp_client_listener.cc
// Code has been simplified and whitespace has been added for readability.

TransportAdapter::Error TcpClientListener::StartListening() {
 if (started_) {
  LOG("TransportAdapter::BAD_STATE. Listener has already been started");
  return TransportAdapter::BAD_STATE;
 }

 if (!interface_listener_->Start()) {
  return TransportAdapter::FAIL;
 }

 if (!IsListeningOnSpecificInterface()) {
  TransportAdapter::Error ret = StartListeningThread();
  if (TransportAdapter::OK != ret) {
   LOG("Tcp client listener thread start failed");
   interface_listener_->Stop();
   return ret;
  }
 }

 started_ = true;
 LOG("Tcp client listener has started successfully");
 return TransportAdapter::OK;
}
```

The next set of functions to implement are StartListening() and ResumeListening(), which are fairly similar, both setting `started_` to true when they are ready to send and receive data. In the case of the TCP Client Listener, ResumeListening initializes the interface listener, and starts the listening thread. StartListening follows a very similar pattern of behavior but calls Start() on the interface listener instead of Init().

```
// tcp_client_listener.cc
// Code has been simplified and whitespace has been added for readability.

TransportAdapter::Error TcpClientListener::StopListening() {
  if (!started_) {
    LOG("TcpClientListener is not running now");
    return TransportAdapter::BAD_STATE;
  }

  interface_listener_->Stop();

  StopListeningThread();

  started_ = false;
  LOG("Tcp client listener was stopped successfully");
  return TransportAdapter::OK;
}
```

The StopListening() and SuspendListening() functions do about the opposite, both stop the TCP Client Listener delegate thread and then set `started_` to false. The difference between the two functions on the TCP Client Listener is that StopListening() will also stop the Platform Specific Network Interface Listener's delegate thread. When SuspendListening() is called, SDL will not be able to create new connections, but existing connections are still able to communicate data. StopListening() will also kill communication with existing connections.

SERVER CONNECTION FACTORY

The Server Connection Factory has the method CreateConnection() which, provided with a device UID and application handle, creates a connection to the application, and then should call ConnectionCreated() on the Transport Adapter.

```
// bluetooth_connection_factory.cc
// Code has been simplified and whitespace has been added for readability.

TransportAdapter::Error BluetoothConnectionFactory::CreateConnection(
    const DeviceUID& device_uid, const ApplicationHandle& app_handle) {
  auto connection = std::make_shared<BluetoothSocketConnection>(
        device_uid, app_handle, controller_);
  controller_->ConnectionCreated(connection, device_uid, app_handle);

  TransportAdapter::Error error = connection->Start();
  if (TransportAdapter::OK != error) {
    LOG("Bluetooth connection::Start() failed with error: "
            << error);
  }

  return error;
}
```

DEVICE SCANNER

The TCP Transport Adapter does not use a device scanner because it waits for incoming connections. We will use the Bluetooth Transport Adapter's Device Scanner as an example here.

The Init() function is called once and is responsible for preparing for the life-cycle of your device scanner. Here, the bluetooth device scanner will start the device scanner worker thread. This worker thread will either scan for devices repeatedly or only when requested via a conditional variable. This behavior is determined by the second and third parameters to the constructor, a boolean `auto_repeat_search` and an integer `auto_repeat_pause_sec`. If `auto_repeat_search` is set to false, the device scanner will only scan when instructed to, otherwise it will scan every `auto_repeat_pause_sec` seconds.

```
// bluetooth_device_scanner.cc
// Code has been simplified and whitespace has been added for readability.

void BluetoothDeviceScanner::Terminate() {
  shutdown_requested_ = true;

  if (thread_) {
    {
      sync_primitives::AutoLock auto_lock(device_scan_requested_lock_);
      device_scan_requested_ = false;
      device_scan_requested_cv_.NotifyOne();
    }

    LOG("Waiting for bluetooth device scanner thread termination");
    thread_->stop();
    LOG("Bluetooth device scanner thread stopped");
  }
}
```

The Terminate() function is called when Core begins shutting down. It will be responsible for telling the worker thread to finish up.

In the case of the bluetooth device scanner, the destructor will join the thread that was started in Init() and cleanup after it.

```
// bluetooth_device_scanner.cc
// Code has been simplified and whitespace has been added for readability.

TransportAdapter::Error BluetoothDeviceScanner::Scan() {
  if (!IsInitialised() || shutdown_requested_) {
    LOG("BAD_STATE");
    return TransportAdapter::BAD_STATE;
  }

  if (auto_repeat_pause_sec_ == 0) {
    return TransportAdapter::OK;
  }

  sync_primitives::AutoLock auto_lock(device_scan_requested_lock_);
  if (!device_scan_requested_) {
    LOG("Requesting device Scan");
    device_scan_requested_ = true;
    device_scan_requested_cv_.NotifyOne();
  } else {
    return TransportAdapter::BAD_STATE;
  }

  return TransportAdapter::OK;
}
```

```
// bluetooth_device_scanner.cc
// Code has been simplified and whitespace has been added for readability.

void BluetoothDeviceScanner::UpdateTotalDeviceList() {
  std::vector<Device*> devices;
  devices.insert(devices.end(),
    paired_devices_with_sdl_.begin(), paired_devices_with_sdl_.end());
  devices.insert(devices.end(),
    found_devices_with_sdl_.begin(), found_devices_with_sdl_.end());

  controller_->SearchDeviceDone(devices);
}
```

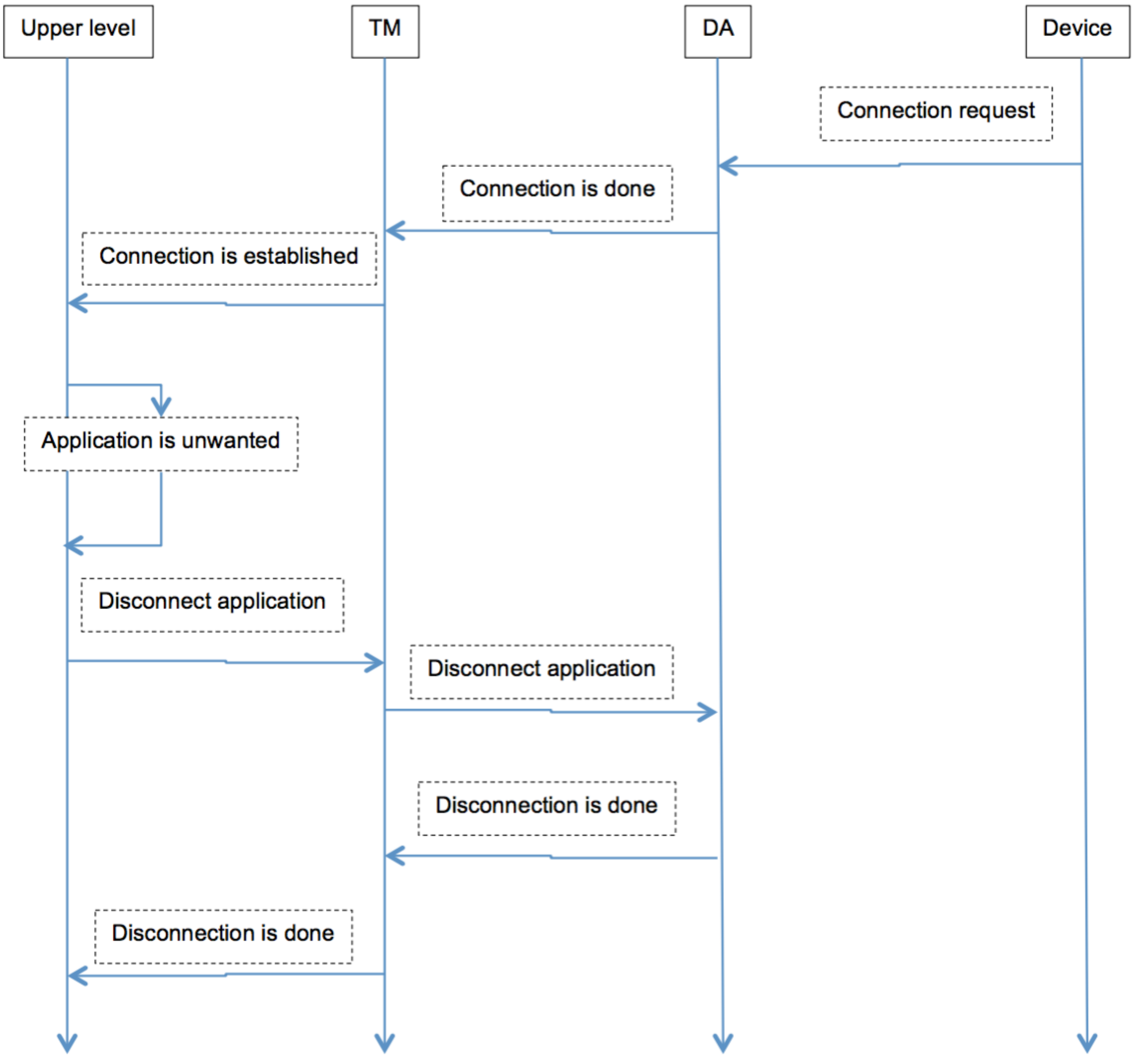
The Scan() function returns an error code, not the actual results of the scan. When the scanning is complete, all devices (existing and newly found) will be passed to the Transport Adapter via the function SearchDeviceDone(). In the Bluetooth device scanner, the Scan() function will signal to the scanning thread that a device scan was requested.

Operation Examples

SEQUENCE DIAGRAM

New Device Connection

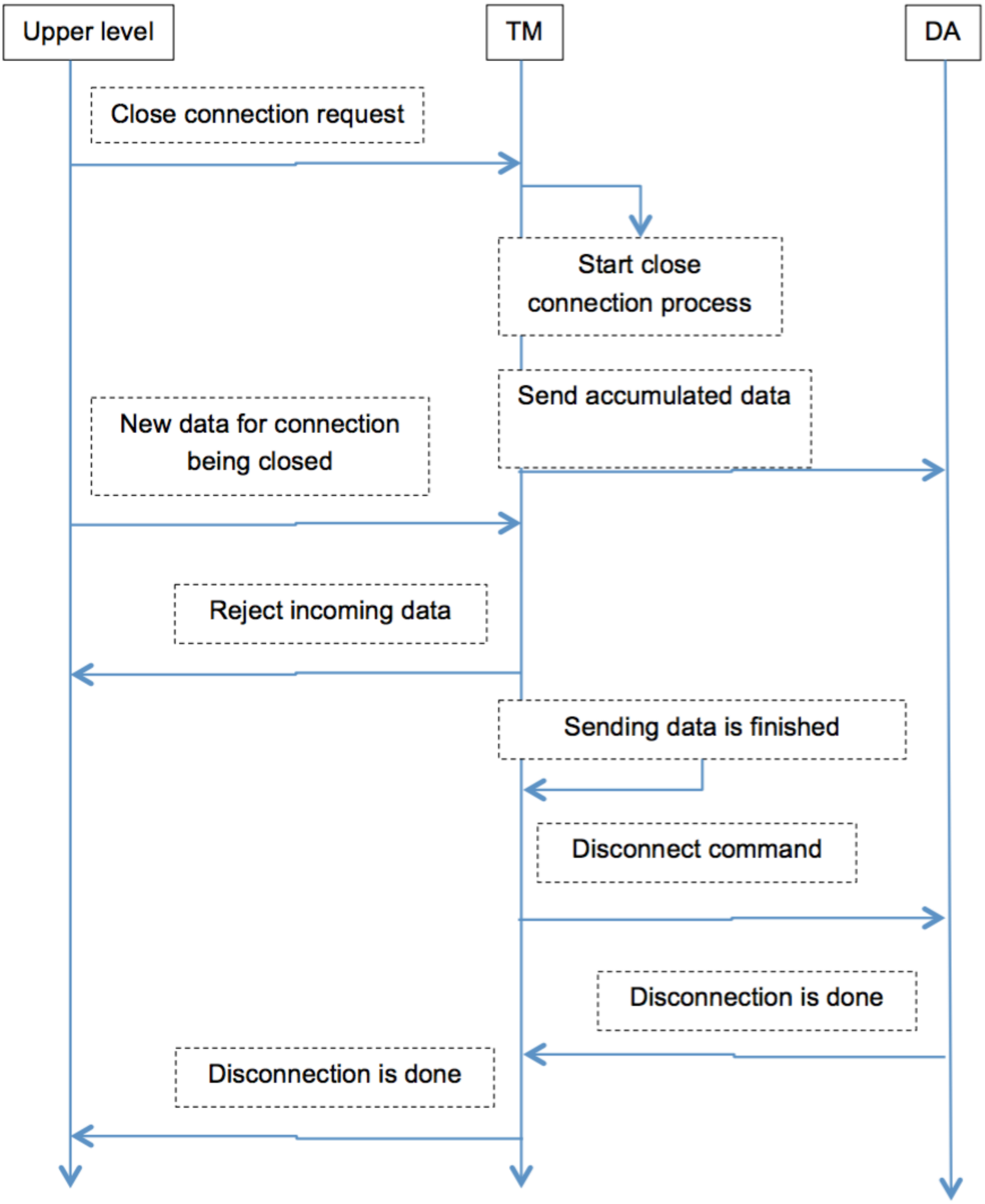
View Diagram



SEQUENCE DIAGRAM

Connection Close Command

View Diagram



Creating a Connection

Core as the Server

Creating a connection with Core acting as the server means that the connection is initiated by a device trying to connect to Core. In the case of the TCP Transport Adapter, this all begins with a device connecting to Core on port 12345. The TCP Connection Listener's loop waits for a new connection to its socket before adding that device to the device list (if it doesn't already exist) and adding the new application to the app list. Once the application has registered and the HMI has received the updated app list, selecting the TCP application in the HMI shall prompt it to activate.

Core as the Client

Creating a connection with Core acting as the client means that the connection is initiated by Core. This means that Core must know in advance how to create the connection. In the case of Cloud applications, their endpoints and names are stored in the policy table enabling them to immediately be included in the app list. When a user activates an application in the HMI, Core will open a web socket connection to the endpoint defined in the policy table and the app may start the RPC service.

Sending and Receiving Data

Sending

Sending data to a device is initiated by the `SendMessageToMobile` method on the RPC Service. This method will post the message to the Protocol Handler and end up in `SendMessageToMobileApp`. This method will, depending on the size of the message, call `SendSingleFrameMessage` or `SendMultiFrameMessage` which will place the messages in the *messages to mobile* queue. Another thread within the Protocol Handler processes messages from this queue and eventually passes them to `SendMessageToDevice` on the transport manager. This again adds the messages to a queue that another thread on the Transport Manager drains, passing the message to the Transport Adapter corresponding to the active Connection. Finally, `SendData` on the Transport Adapter which does the actual sending of the raw data. It is good to note that in some of the existing Transport Adapters, the code to actually transmit data is in `SendData` on the Connection object, and the Transport Adapter `SendData` call will be forwarded to `SendData` on the Connection object.

Receiving

The code to receive data will vary depending on the method of transport; in the case of the TCP Transport Adapter, the Socket Connection thread loops checking if data has been sent to its socket before calling `recv` and converting the read buffer to a raw message. When a Transport Adapter finishes receiving incoming message(s) it will emit the event OnReceivedDone. This event will be propagated to the Transport Manager Listeners including the Protocol Handler who will add the message(s) to the *messages from mobile* queue which is processed by another thread within the Protocol Handler.

Events

Transport Adapter Events

These events are generated by a Transport Adapter and forwarded to a Transport Adapter Listener who will in turn post the event to the Transport Manager who will finally raise the event to the Transport Manager Listeners. The only exception is OnSendFail, which is not forwarded from the Transport Manager to the Transport Manager Listeners.

OnSearchDone

Indicates that a device search has completed. In the case of the USB Transport Adapter, this event is emitted by the Device Scanner after a scan has completed.

OnSearchFail

Indicates that a device search has encountered an error. In the case of the Bluetooth Transport Adapter, this event is emitted by the Device Scanner when Core fails to correctly interact with the bluetooth hardware.

OnDeviceListUpdated

Indicates that the list of connected devices has been updated. This event will be emitted when `AddDevice`, `RemoveDevice` or `SearchDeviceDone` is called on the transport adapter.

OnFindNewApplicationsRequest

Indicates that SDL Core should begin to check for new applications on newly connected devices. This event is emitted by the Bluetooth Device Scanner once it has connected to new devices and updated the device list.

OnConnectDone

Indicates that a connection has been established. The Transport Manager will then add the connection to the connection list if it has not already been added. In the case of the cloud websocket transport adapter, this event is emitted once the connection handshake is completed.

OnDisconnectDone

Indicates that disconnecting from an application has completed and prompts the Transport Manager to remove the connection from the connection list. In the case of the websocket Transport Adapter, this event is emitted after the delegate threads for the connection have been stopped.

OnSendDone

Indicates that a Transport Adapter has finished sending the messages in its queue. Upon receipt of this event the Transport Manager will check if the connection is slated for shutdown and disconnect it if so.

OnSendFail

Indicates that a Transport Adapter failed to send a message properly. This could prompt the transport manager to take action that would reconcile the errors. This event is not currently raised to the Transport Manager Listeners.

OnReceivedDone

Indicates that a Transport Adapter has successfully received a message. The received data is eventually passed to Transport Manager Listeners which will process that data.

OnReceivedFail

Indicates that a Transport Adapter has failed to properly receive a message.

OnUnexpectedDisconnect

Indicates that a device has been unexpectedly disconnected. This event could be emitted in the case of a device being disconnected or a connection being aborted. This event will prompt the Transport Manager to remove the disconnected connection.

OnTransportSwitchRequested

Indicates that a transport switch has been requested. This will prompt the Transport Manager to begin transport switching.

OnTransportConfigUpdated

Indicates that the Transport Config has been updated. This will prompt the Protocol Handler (a Transport Manager Listener) to check for updates to things like the TCP listening address and port.

OnConnectPending

Indicates that a connection is pending. The Transport Manager will then add the connection to the connection list if it has not already been added. In the case of the cloud websocket transport adapter, this event is emitted once a connection configuration is known to SDL Core, but before the connection is actually established.

OnConnectionStatusUpdated

Indicates that the status of one or more connections has been updated, and SDL Core should send an `UpdateAppList` RPC to the HMI. This will be emitted by the Transport Adapter during device connection and also when a device is disconnected.

Transport Manager Listener Events

These events are created in the Transport Manager and only raised to the Transport Manager Listeners.

OnDeviceAdded and OnDeviceRemoved

These two events are fired when `UpdateDeviceList` is called on the Transport Manager. One OnDeviceAdded event will be dispatched for each new device in the list and one OnDeviceRemoved will be dispatched for each device that is no longer in the list.

OnDeviceFound

When `UpdateDeviceMapping` is called on the Transport Manager, it ensures all devices from a Transport Adapter's device list are accounted for in the device to adapter map. OnDeviceFound will be raised for any new devices that weren't previously in the device to adapter map.

OnDeviceSwitchingStart

When `TryDeviceSwitch` is called on the Transport Manager following an OnTransportSwitchRequested event, the OnDeviceSwitchingStart event is raised with both the bluetooth and USB device UIDs.

Resume Controller

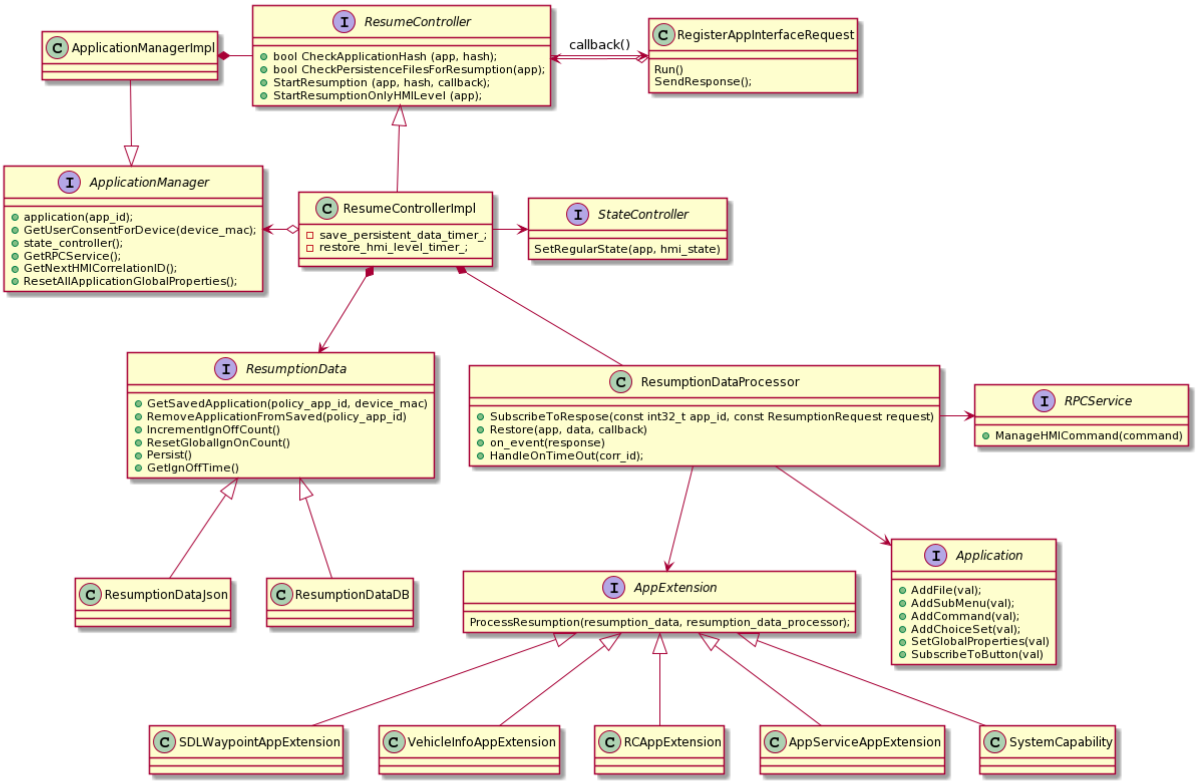
This page will describe internal structure and detailed design of Resume controller

SEQUENCE DIAGRAM

Figure 1: ResumeController Overview

View Diagram

ResumeController Big Picture


ResumeController
bool CheckApplicationHash (app, hash);
bool CheckPersistenceFilesForResumption(app);
StartResumption (app, hash, callback);
StartResumptionOnlyHMILevel (app);
ApplicationManagerImpl
callback()
RegisterAppInterfaceRequest
Run()
SendResponse();
ApplicationManager
application(app_id);
GetUserConsentForDevice(device_mac);
state_controller();
GetRPCService();
GetNextHMICorrelationID();
ResetAllApplicationGlobalProperties();
ResumeControllerImpl
save_persistent_data_timer_;
restore_hmi_level_timer_;
StateController
SetRegularState(app, hmi_state)
ResumptionData
GetSavedApplication(policy_app_id, device_mac)
RemoveApplicationFromSaved(policy_app_id)
IncrementIgnOffCount()
ResetGlobalIgnOnCount()
Persist()
GetIgnOffTime()
ResumptionDataProcessor
SubscribeToRespose(const int32_t app_id, const ResumptionRequest request)
Restore(app, data, callback)
on_event(response)
HandleOnTimeOut(corr_id);
RPCService
ManageHMICommand(command)
ResumptionDataJson
ResumptionDataDB
AppExtension
ProcessResumption(resumption_data, resumption_data_processor);
Application
AddFile(val);
AddSubMenu(val);
AddCommand(val);
AddChoiceSet(val);
SetGlobalProperties(val)
SubscribeToButton(val)
SDLWaypointAppExtension
VehicleInfoAppExtension
RCAppExtension
AppServiceAppExtension
SystemCapability

NOTE

Classes named like *Impl only *represent* implementations of the abstract sub classes and may not be named the same in the SDL Core project.
UML Refresher

- Aggregation: Solid line with open diamond
- Composition: Solid line with filled diamond
- Inheritance: Dotted line with open arrow
- Dependency: Dotted line with two prong arrow

Resume Controller

The resume controller's responsibility is to handle the resumption responsibilities of SDL. There are 2 resumption types :

- HMI state resumption
- Data resumption

The resume controller does both.

HMI state resumption

In the case of unexpected disconnect SDL should store an application's HMI state for the next 3 ignition cycles.
On next application registration SDL should restore last saved application HMI state.

ResumptionData is responsible for application data restoring.
ResumeCtrlImpl is responsible for HMI state restoring.

ResumeCtrlImpl will remove application hmi_state info from resumption data after 3 ignition cycles.
On each shutdown ResumeCtrlImpl will increment `ign_off_count` value for each application.

On App registration `ResumeCtrl::StartResumptionOnlyHMILevel` or `ResumeCtrlImpl::StartResumption` will put application in a queue for resumption.
Internal timer in ResumeCtrlImpl will restore application hmi_state in several seconds (configured by `ApplicationManagerSettings::app_resuming_timeout`)
In the case where another application has already registered, the StateController will take care of resolving any HMI state conflicts.

Data resumption

SDL restores application data if an application sends the appropriate `hashID` in the RegisterAppInterface request. This hash updates after each data change.
SDL stores resumption data either in json or in database, this option is configurable via INI file `UseDBForResumption=false` field in `[Resumption]` section.

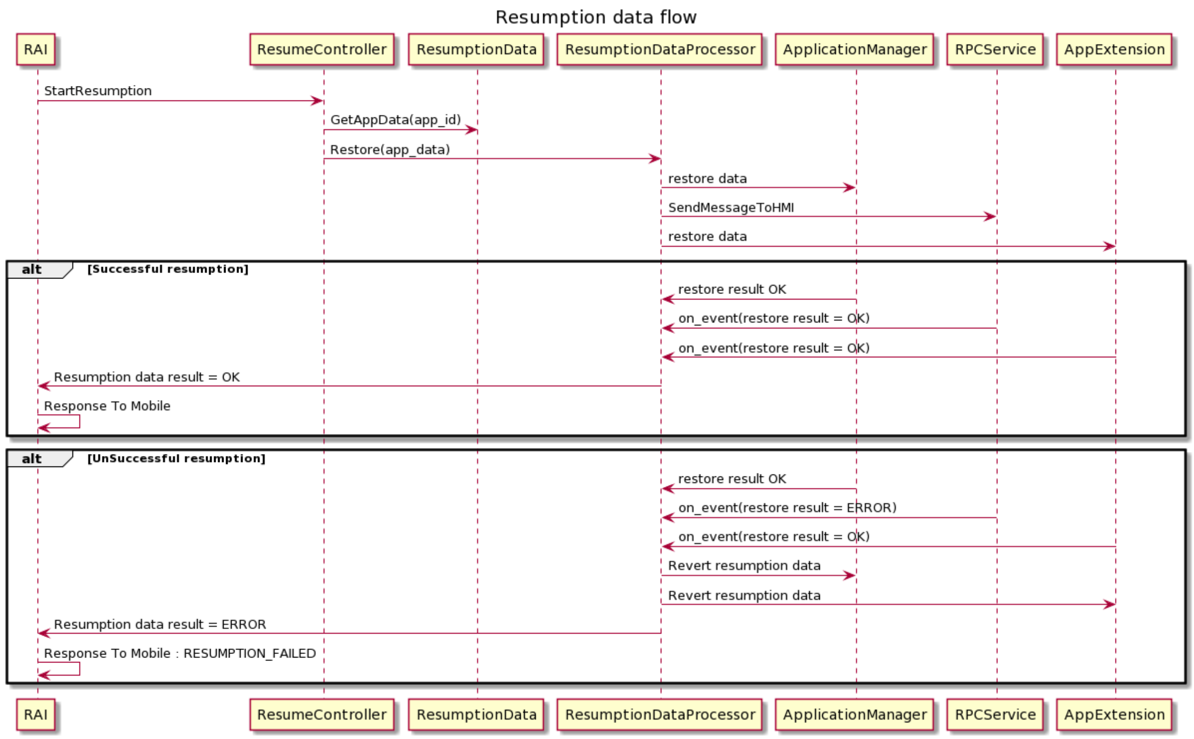
ResumeControllerImpl requests app data from `ResumptionData` class and provides it to `ResumptionDataProcessor`

`ResumptionDataProcessor` is responsible for restoring application data and provides the result to RegisterAppInterface via a callback.

SEQUENCE DIAGRAM

Figure 2: Resumption data sequence Overview

View Diagram



ResumptionData

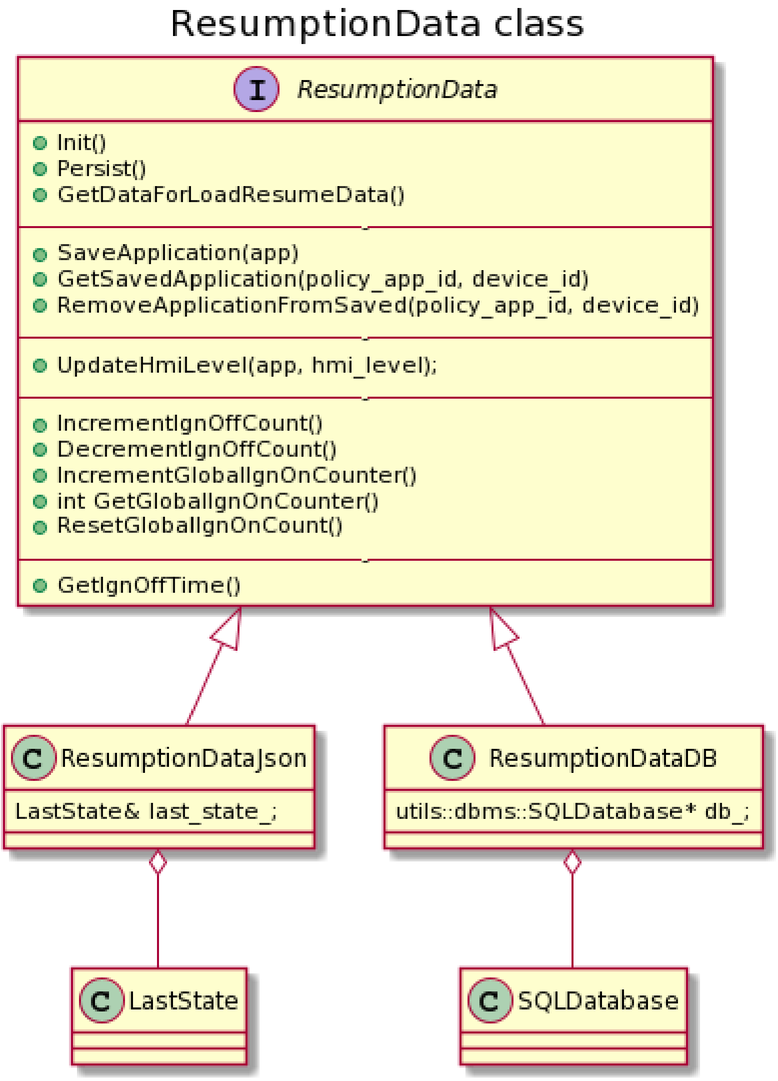
`ResumptionData` class is used to represent resumption data agnostic to data storage. `ResumptionData` provides app resumption data in the Smart Object representation.

SEQUENCE DIAGRAM

Figure 2: Resumption data classes

View Diagram

View Diagram


ResumptionData class
I ResumptionData
Init()
Persist()
GetDataForLoadResumeData()
SaveApplication(app)
GetSavedApplication(policy_app_id, device_id)
RemoveApplicationFromSaved(policy_app_id, device_id)
UpdateHmiLevel(app, hmi_level);
IncrementIgnOffCount()
DecrementIgnOffCount()
IncrementGlobalIgnOnCounter()
int GetGlobalIgnOnCounter()
ResetGlobalIgnOnCount()
GetIgnOffTime()
C ResumptionDataJson
LastState& last_state_;
C ResumptionDataDB
utils::dbms::SQLDatabase* db_;
C LastState
C SQLDatabase

There are 2 implementations of resumption data :
* ResumptionDataJson
* ResumptionDataDB

ResumptionData does not contain active components : timers, reactions, callbacks, etc ...
It is responsible for data storage.

ResumptionDataProcessor

ResumptionDataProcessor is responsible for restoring resumption data and tracking its status.

Main public function for resumptions is `ResumptionDataProcessor::Restore` :

```
/**
* @brief Running resumption data process from saved_app to application.
* @param application Application which will be resumed
* @param saved_app Application specific section from backup file
* @param callback Function signature to be called when
* data resumption will be finished
*/
void Restore(app_mngr::ApplicationSharedPtr application,
             smart_objects::SmartObject& saved_app,
             ResumeCtrl::ResumptionCallBack callback);
```

`ResumeCtrl::ResumptionCallBack callback` is a function that should be called after data resumption :

```
typedef std::function<void(mobile_apis::Result::eType result_code,
                           const std::string& info)> ResumptionCallBack;
```

- Some resumption data should be restored in the `Application` class itself.
- Some resumption data should be stored in plugins : ApplicationExtensions.
- Some resumption data requires sending HMI requests.

`ResumptionDataProcessor` is inherited from `EventObserver` to track responses.

If all responses are successful `ResumptionDataProcessor` will call `callback(SUCCESS)`

If some of the data failed to restore, `ResumptionDataProcessor` will revert already restored data and call `callback(ERROR_CODE, info)` .

The requirements are available in proposal 0190: Handle response from HMI during resumption data

RegisterAppInterface will wait for the callback to send a response to a mobile application.

AppExtension

Application extension contains following methods for resumption :

```
/**
 * @brief SaveResumptionData method called by SDL when it saves resumption
 * data.
 * @param resumption_data data reference to data, that will be appended by
 * plugin
 */
virtual void SaveResumptionData(
    smart_objects::SmartObject& resumption_data) = 0;

/**
 * @brief ProcessResumption Method called by SDL during resumption.
 * @param resumption_data list of resumption data
 */
virtual void ProcessResumption(
    const smart_objects::SmartObject& resumption_data) = 0;

/**
 * @brief RevertResumption Method called by SDL during revert resumption.
 * @param subscriptions Subscriptions from which must discard
 */
virtual void RevertResumption(
    const smart_objects::SmartObject& subscriptions) = 0;
```

Only an application's extension have an access to active data, data send and data revert process.
Each application extension uses its own plugin to manipulate with functionality.

`SaveResumptionData` will fill passed `resumption_data` for saving to `ResumptionData`.

Example from VehicleInfoAppExtension:

```
SDLRPCPlugin& plugin_;
...
void VehicleInfoAppExtension::SaveResumptionData(
    smart_objects::SmartObject& resumption_data) {
  resumption_data[strings::application_vehicle_info] =
      smart_objects::SmartObject(smart_objects::SmartType_Array);
  int i = 0;
  for (const auto& subscription : subscribed_data_) {
    resumption_data[strings::application_vehicle_info][i++] = subscription;
  }
}
```

`ProcessResumption` will send appropriate HMI requests, and change internal SDL state according to provided `resumption_data`. All HMI responses will be transferred to `ResumptionDataProcessor`

Example from SDLWaypointAppExtension:

```
SDLRPCPlugin& plugin_;
...
void SDLWaypointAppExtension::ProcessResumption(
    const smart_objects::SmartObject& saved_app) {
  ...
  const bool subscribed_for_way_points =
      saved_app[strings::subscribed_for_way_points].asBool();
  if (subscribed_for_way_points_so) {
    plugin_.ProcessResumptionSubscription(app_, *this);
  }
}
```

On each request sent to HMI Plugin will call `resumption_data_processor->SubscribeOnResponse`.
This will inform ResumptionDataProcessor that it should wait for a response before finishing resumption and sending RAI response to mobile.

`RevertResumption` will send the appropriate HMI requests to revert provided `subscriptions`.

Resumption of Subscriptions

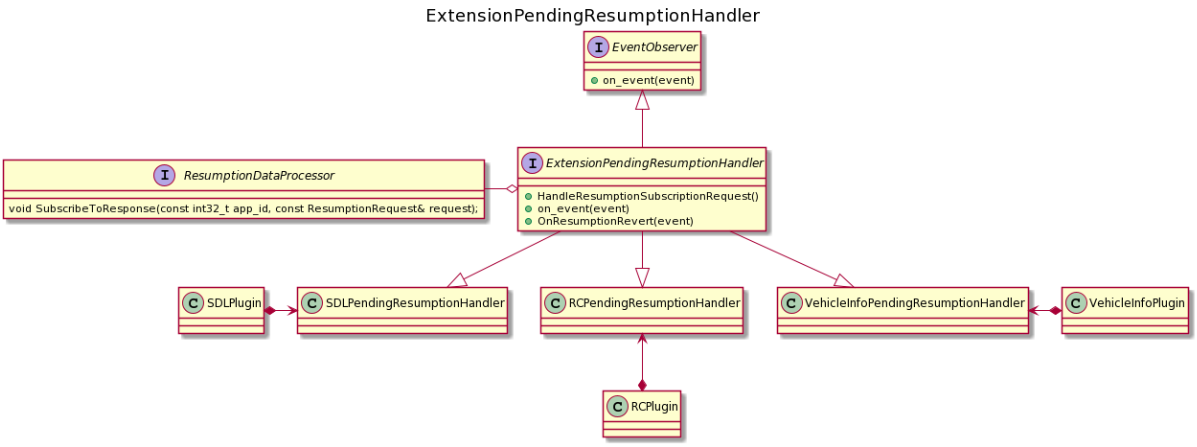
If multiple applications are trying to restore the same subscription, SDL should send the only first subscription to HMI. If the first subscription was failed and the application received `RESUME_FAILED` result code, for the second application SDL should also try to restore the subscription.

For the waiting subscription result, SDL uses the `ExtensionPendingResumptionHandler` class.
Each plugin contains its own ExtensionPendingResumptionHandler for subscriptions resumption.

SEQUENCE DIAGRAM

ExtensionPendingResumptionHandler overview

View Diagram


ExtensionPendingResumptionHandler
I EventObserver
on_event(event)
I ResumptionDataProcessor
void SubscribeToResponse(const int32_t app_id, const ResumptionRequest& request);
I ExtensionPendingResumptionHandler
HandleResumptionSubscriptionRequest()
on_event(event)
OnResumptionRevert(event)
C SDLPlugin
C SDLPendingResumptionHandler
C RCPendingResumptionHandler
C VehicleInfoPendingResumptionHandler
C VehicleInfoPlugin
C RCPlugin

For subscriptions resumption plugin calls `ExtensionPendingResumptionHandler::HandleResumptionSubscriptionRequest(app_extension, application)`

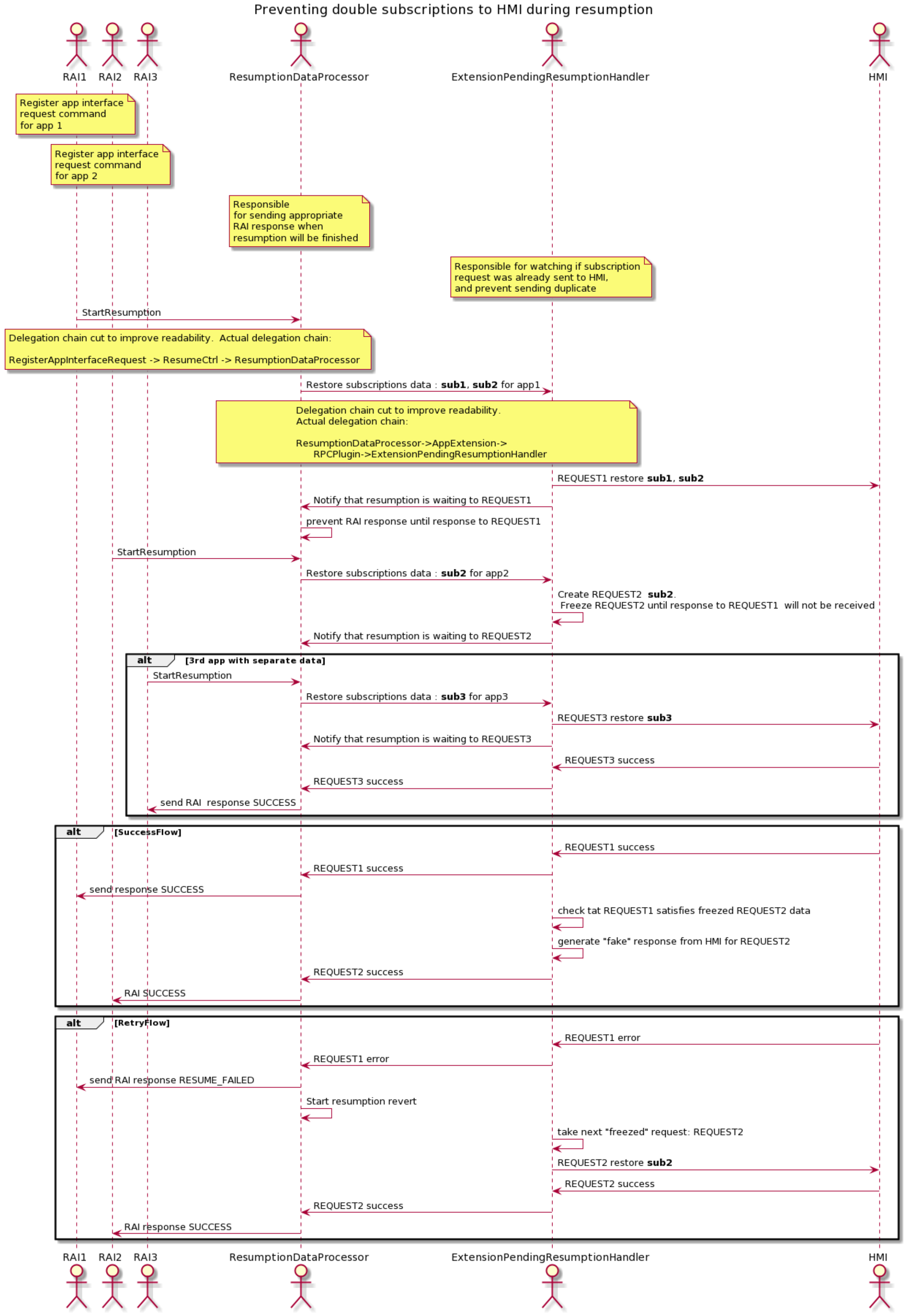
`ExtensionPendingResumptionHandler` sends requests to HMI for all subscriptions available in `app_extension` and tracks responses with the `on_event` method inherited from `EventObserver`.

In the case some subscription request to the HMI was already sent but the response was not received yet, `ExtensionPendingResumptionHandler` will not send an additional request to HMI but store internally that appropriate subscription resumption is "frozen". When the response is received from the HMI, SDL will manage both resumptions according to response data.
For "frozen" resumptions ExtensionPendingResumptionHandler will raise an event so that `ResumeDataProcessor` will receive this event and understand it as response from HMI.

SEQUENCE DIAGRAM

Subscriptions restore sequence :

View Diagram


Preventing double subscriptions to HMI during resumption
RAI1
RAI2
RAI3
ResumptionDataProcessor
ExtensionPendingResumptionHandler
HMI
Register app interface request command for app 1
Register app interface request command for app 2
Responsible for sending appropriate RAI response when resumption will be finished
Responsible for watching if subscription request was already sent to HMI, and prevent sending duplicate
StartResumption
Delegation chain cut to improve readability. Actual delegation chain:
RegisterAppInterfaceRequest -> ResumeCtrl -> ResumptionDataProcessor
Restore subscriptions data : sub1, sub2 for app1
Delegation chain cut to improve readability.
Actual delegation chain:
ResumptionDataProcessor->AppExtension->
RPCPlugin->ExtensionPendingResumptionHandler
REQUEST1 restore sub1, sub2
Notify that resumption is waiting to REQUEST1
prevent RAI response until response to REQUEST1
StartResumption
Restore subscriptions data : sub2 for app2
Create REQUEST2 sub2.
Freeze REQUEST2 until response to REQUEST1 will not be received
Notify that resumption is waiting to REQUEST2
alt
[3rd app with separate data]
StartResumption
Restore subscriptions data : sub3 for app3
REQUEST3 restore sub3
Notify that resumption is waiting to REQUEST3
REQUEST3 success
REQUEST3 success
send RAI response SUCCESS
alt
[SuccessFlow]
REQUEST1 success
REQUEST1 success
send response SUCCESS
check tat REQUEST1 satisfies freezed REQUEST2 data
generate "fake" response from HMI for REQUEST2
REQUEST2 success
RAI SUCCESS
alt
[RetryFlow]
REQUEST1 error
REQUEST1 error
send RAI response RESUME_FAILED
Start resumption revert
take next "freezed" request: REQUEST2
REQUEST2 restore sub2
REQUEST2 success
REQUEST2 success
RAI response SUCCESS
RAI1
RAI2
RAI3
ResumptionDataProcessor
ExtensionPendingResumptionHandler
HMI

OnResumptionRevert is used to trigger the next frozen resumption if no requests are currently waiting for a response.

Security Manager Guide

What is the Security Manager?

Secure communication in SDL Core is provided by the Security Manager interface, implemented in Core as the SecurityManagerImpl.

Under the hood the Security Manager uses OpenSSL to complete handshakes and encrypt/decrypt data. All OpenSSL operations are abstracted away by the Crypto Manager - which provides a factory for SSL Context objects, the ability to update certificates from Certificate Authorities and SSL error reporting. The SSL Context objects created by the Crypto Manager can be used to establish SSL connections, as well as to encrypt or decrypt data. Events within the Security Manager will be propagated to any Security Manager Listeners that register with the Security Manager. The Protocol Handler implementation in Core implements the SecurityManagerListener interface in order to send protocol layer responses to handshakes.

What events are generated?

ONHANDSHAKEDONE

When a handshake has been completed, the `OnHandshakeDone` is called on each registered `SecurityManagerListener`. This function is called with two parameters: the connection key, and a HandshakeResult enum value (one of Fail, Success, CertExpired, etc.)

ONCERTIFICATEUPDATEREQUIRED

When a handshake is initiated and the connection does not have a valid certificate, `OnCertificateUpdateRequired` will be called on each registered SecurityManagerListener. This could be either because no certificate was supplied for the connection or because the certificate is expired.

ONGETSYSTEMTIMEFAILED

If the Security Manager fails to get the system time from the HMI, `OnGetSystemTimeFailed` will be called on each registered SecurityManagerListener.

ONCERTDECRYPTFAILED

If the Security Manager fails to decrypt a certificate, `OnCertDecryptFailed` will be called on each registered SecurityManagerListener.

ONPTUFAILED

If a PTU fails, each registered `SecurityManagerListener` is notified with the OnPTUFailed event.

Sequence Diagrams

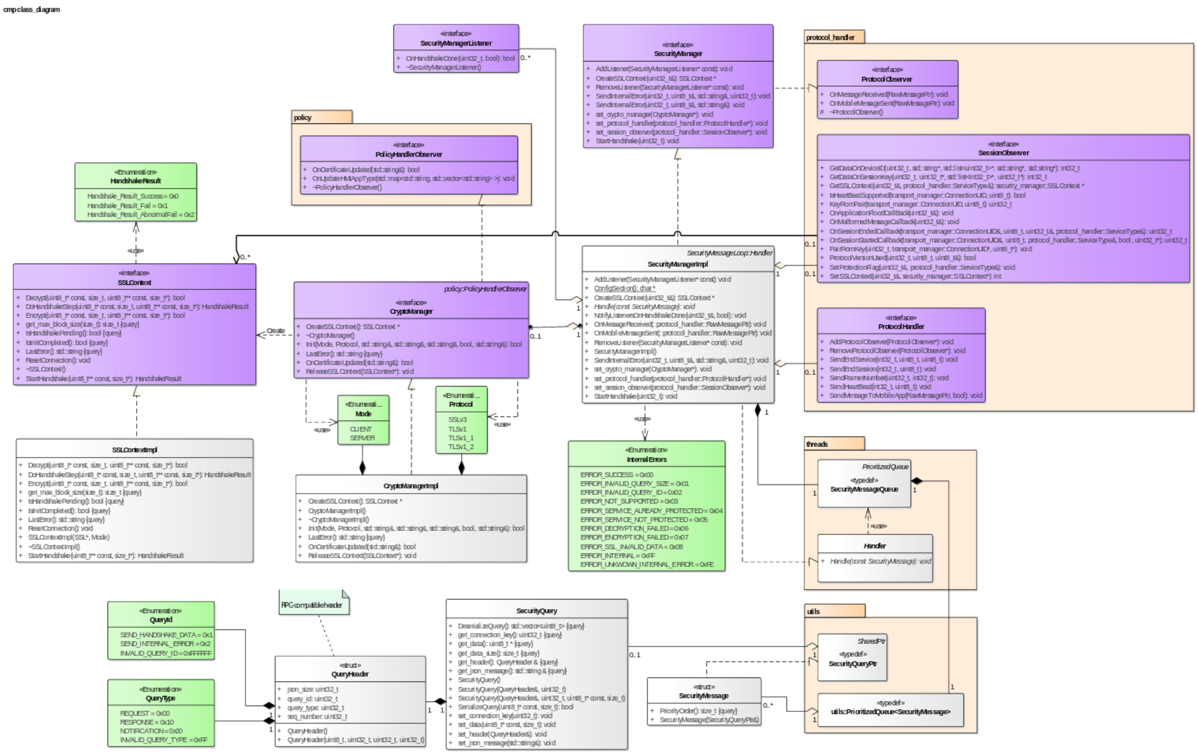
SECURITY MANAGER COMPONENT HIERARCHY

To further understand the relationship between the components of the security manager, please take a look at this UML diagram.

SEQUENCE DIAGRAM

Security Manager UML

View Diagram


cmp class_diagram
«interface» SecurityManagerListener
+ OnHandshakeDone(uint32_t, bool): bool
+ ~SecurityManagerListener()
0..*
«interface» SecurityManager
+ AddListener(SecurityManagerListener* const): void
+ CreateSSLContext(uint32_t&): SSLContext *
+ RemoveListener(SecurityManagerListener* const): void
+ SendInternalError(uint32_t, uint8_t&, std::string&, uint32_t): void
+ SendInternalError(uint32_t, uint8_t&, std::string&): void
+ set_crypto_manager(CryptoManager*): void
+ set_protocol_handler(protocol_handler::ProtocolHandler*): void
+ set_session_observer(protocol_handler::SessionObserver*): void
+ StartHandshake(uint32_t): void
protocol_handler
«interface» ProtocolObserver
+ OnMessageReceived(RawMessagePtr): void
+ OnMobileMessageSent(RawMessagePtr): void
# ~ProtocolObserver()
«interface» SessionObserver
+ GetDataOnDeviceID(uint32_t, std::string*, std::list<uint32_t>*, std::string*, std::string*): int32_t
+ GetDataOnSessionKey(uint32_t, uint32_t*, std::list<int32_t>*, uint32_t*): int32_t
+ GetSSLContext(uint32_t&, protocol_handler::ServiceType&): security_manager::SSLContext *
+ IsHeartBeatSupported(transport_manager::ConnectionUID, uint8_t): bool
+ KeyFromPair(transport_manager::ConnectionUID, uint8_t): uint32_t
+ OnApplicationFloodCallBack(uint32_t&): void
+ OnMalformedMessageCallback(uint32_t&): void
+ OnSessionEndedCallback(transport_manager::ConnectionUID&, uint8_t, uint32_t&, protocol_handler::ServiceType&): uint32_t
+ OnSessionStartedCallback(transport_manager::ConnectionUID&, uint8_t, protocol_handler::ServiceType&, bool, uint32_t*): uint32_t
+ PairFromKey(uint32_t, transport_manager::ConnectionUID*, uint8_t*): void
+ ProtocolVersionUsed(uint32_t, uint8_t, uint8_t&): bool
+ SetProtectionFlag(uint32_t&, protocol_handler::ServiceType&): void
+ SetSSLContext(uint32_t&, security_manager::SSLContext*): int
«interface» ProtocolHandler
+ AddProtocolObserver(ProtocolObserver*): void
+ RemoveProtocolObserver(ProtocolObserver*): void
+ SendEndService(int32_t, uint8_t, uint8_t): void
+ SendEndSession(int32_t, uint8_t): void
+ SendFramesNumber(uint32_t, int32_t): void
+ SendHeartBeat(int32_t, uint8_t): void
+ SendMessageToMobileApp(RawMessagePtr, bool): void
policy
«interface» PolicyHandlerObserver
+ OnCertificateUpdated(std::string&): bool
+ OnUpdateHMIAppType(std::map<std::string, std::vector<std::string> >): void
+ ~PolicyHandlerObserver()
«Enumeration» HandshakeResult
Handshake_Result_Success = 0x0
Handshake_Result_Fail = 0x1
Handshake_Result_AbnormalFail = 0x2
«use»
0..*
«interface» SSLContext
+ Decrypt(uint8_t* const, size_t, uint8_t** const, size_t*): bool
+ DoHandshakeStep(uint8_t* const, size_t, uint8_t** const, size_t*): HandshakeResult
+ Encrypt(uint8_t* const, size_t, uint8_t** const, size_t*): bool
+ get_max_block_size(size_t): size_t {query}
+ IsHandshakePending(): bool {query}
+ IsInitCompleted(): bool {query}
+ LastError(): std::string {query}
+ ResetConnection(): void
+ ~SSLContext()
+ StartHandshake(uint8_t** const, size_t*): HandshakeResult
Create
policy::PolicyHandlerObserver
«interface» CryptoManager
+ CreateSSLContext(): SSLContext *
+ ~CryptoManager()
+ Init(Mode, Protocol, std::string&, std::string&, std::string&, bool, std::string&): bool
+ LastError(): std::string {query}
+ OnCertificateUpdated(std::string&): bool
+ ReleaseSSLContext(SSLContext*): void
0..1
SecurityMessageLoop::Handler
SecurityManagerImpl
+ AddListener(SecurityManagerListener* const): void
+ ConfigSection(): char *
+ CreateSSLContext(uint32_t&): SSLContext *
+ Handle(const SecurityMessage): void
+ NotifyListenersOnHandshakeDone(uint32_t&, bool): void
+ OnMessageReceived(::protocol_handler::RawMessagePtr): void
+ OnMobileMessageSent(::protocol_handler::RawMessagePtr): void
+ RemoveListener(SecurityManagerListener* const): void
+ SecurityManagerImpl()
+ SendInternalError(uint32_t, uint8_t&, std::string&, uint32_t): void
+ set_crypto_manager(CryptoManager*): void
+ set_protocol_handler(protocol_handler::ProtocolHandler*): void
+ set_session_observer(protocol_handler::SessionObserver*): void
+ StartHandshake(uint32_t): void
1
0..1
«Enumerati... Mode
CLIENT
SERVER
«use»
«Enumerati... Protocol
SSLv3
TLSv1
TLSv1_1
TLSv1_2
«use»
SSLContextImpl
+ Decrypt(uint8_t* const, size_t, uint8_t** const, size_t*): bool
+ DoHandshakeStep(uint8_t* const, size_t, uint8_t** const, size_t*): HandshakeResult
+ Encrypt(uint8_t* const, size_t, uint8_t** const, size_t*): bool
+ get_max_block_size(size_t): size_t {query}
+ IsHandshakePending(): bool {query}
+ IsInitCompleted(): bool {query}
+ LastError(): std::string {query}
+ ResetConnection(): void
+ SSLContextImpl(SSL*, Mode)
+ ~SSLContextImpl()
+ StartHandshake(uint8_t** const, size_t*): HandshakeResult
CryptoManagerImpl
+ CreateSSLContext(): SSLContext *
+ CryptoManagerImpl()
+ ~CryptoManagerImpl()
+ Init(Mode, Protocol, std::string&, std::string&, std::string&, bool, std::string&): bool
+ LastError(): std::string {query}
+ OnCertificateUpdated(std::string&): bool
+ ReleaseSSLContext(SSLContext*): void
«use»
«Enumeration» InternalErrors
ERROR_SUCCESS = 0x00
ERROR_INVALID_QUERY_SIZE = 0x01
ERROR_INVALID_QUERY_ID = 0x02
ERROR_NOT_SUPPORTED = 0x03
ERROR_SERVICE_ALREADY_PROTECTED = 0x04
ERROR_SERVICE_NOT_PROTECTED = 0x05
ERROR_DECRYPTION_FAILED = 0x06
ERROR_ENCRYPTION_FAILED = 0x07
ERROR_SSL_INVALID_DATA = 0x08
ERROR_INTERNAL = 0xFF
ERROR_UNKWOWN_INTERNAL_ERROR = 0xFE
threads
PrioritizedQueue
«typedef» SecurityMessageQueue
«use»
Handler
+ Handle(const SecurityMessage): void
«Enumeration» QueryId
SEND_HANDSHAKE_DATA = 0x1
SEND_INTERNAL_ERROR = 0x2
INVALID_QUERY_ID = 0xFFFFFF
«Enumeration» QueryType
REQUEST = 0x00
RESPONSE = 0x10
NOTIFICATION = 0x20
INVALID_QUERY_TYPE = 0xFF
RPC-compatible header
«struct» QueryHeader
+ json_size: uint32_t
+ query_id: uint32_t
+ query_type: uint32_t
+ seq_number: uint32_t
+ QueryHeader()
+ QueryHeader(uint8_t, uint32_t, uint32_t, uint32_t)
SecurityQuery
+ DeserializeQuery(): std::vector<uint8_t> {query}
+ get_connection_key(): uint32_t {query}
+ get_data(): uint8_t * {query}
+ get_data_size(): size_t {query}
+ get_header(): QueryHeader & {query}
+ get_json_message(): std::string & {query}
+ SecurityQuery()
+ SecurityQuery(QueryHeader&, uint32_t)
+ SecurityQuery(QueryHeader&, uint32_t, uint8_t* const, size_t)
+ SerializeQuery(uint8_t* const, size_t): bool
+ set_connection_key(uint32_t): void
+ set_data(uint8_t* const, size_t): void
+ set_header(QueryHeader&): void
+ set_json_message(std::string&): void
0..1
«struct» SecurityMessage
+ PriorityOrder(): size_t {query}
+ SecurityMessage(SecurityQueryPtr&)
0..*
utils
SharedPtr
«typedef» SecurityQueryPtr
«typedef» utils::PrioritizedQueue<SecurityMessage>

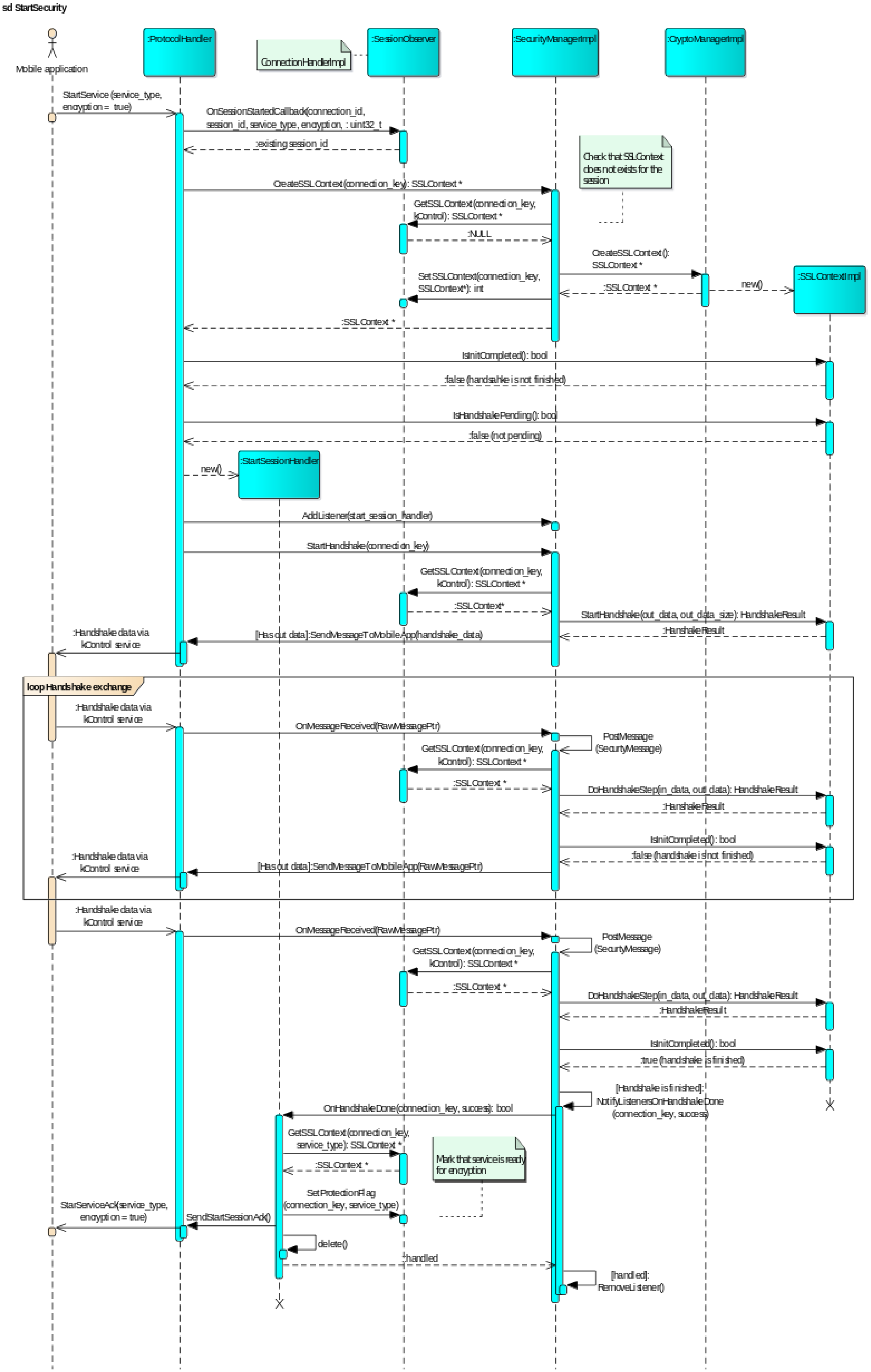
SECURITY MANAGER INITIALIZATION

To understand how the security manager is initialized, please take a look at this flow diagram.

SEQUENCE DIAGRAM

Security Manager Initialization

View Diagram


sd StartSecurity
Mobile application
:ProtocolHandler
ConnectionHandlerImpl
:SessionObserver
:SecurityManagerImpl
:CryptoManagerImpl
StartService (service_type, encryption = true)
OnSessionStartedCallback(connection_id, session_id, service_type, encryption, : uint32_t
:existing session_id
Check that SSLContext does not exists for the session
CreateSSLContext(connection_key): SSLContext *
GetSSLContext(connection_key, kControl): SSLContext *
:NULL
CreateSSLContext(): SSLContext *
:SSLContext *
new()
:SSLContextImpl
SetSSLContext(connection_key, SSLContext*): int
:SSLContext *
IsInitCompleted(): bool
:false (handsahke is not finished)
IsHandshakePending(): bool
:false (not pending)
new()
:StartSessionHandler
AddListener(start_session_handler)
StartHandshake(connection_key)
GetSSLContext(connection_key, kControl): SSLContext *
:SSLContext*
StartHandshake(out_data, out_data_size): HandshakeResult
:HanshakeResult
:Handshake data via kControl service
[Has out data]:SendMessageToMobileApp(handshake_data)
loop Handshake exchange
:Handshake data via kControl service
OnMessageReceived(RawMessagePtr)
PostMessage (SecurityMessage)
GetSSLContext(connection_key, kControl): SSLContext *
:SSLContext *
DoHandshakeStep(in_data, out_data): HandshakeResult
:HanshakeResult
IsInitCompleted(): bool
:false (handshake is not finished)
:Handshake data via kControl service
[Has out data]:SendMessageToMobileApp(RawMessagePtr)
:Handshake data via kControl service
OnMessageReceived(RawMessagePtr)
PostMessage (SecurityMessage)
GetSSLContext(connection_key, kControl): SSLContext *
:SSLContext *
DoHandshakeStep(in_data, out_data): HandshakeResult
:HanshakeResult
IsInitCompleted(): bool
:true (handshake is finished)
[Handshake is finished]: NotifyListenersOnHandshakeDone (connection_key, success)
OnHandshakeDone(connection_key, success): bool
GetSSLContext(connection_key, service_type): SSLContext *
:SSLContext *
Mark that service is ready for encryption
SetProtectionFlag (connection_key, service_type)
StarServiceAck(service_type, encryption = true)
SendStartSessionAck()
delete()
:handled
[handled]: RemoveListener()

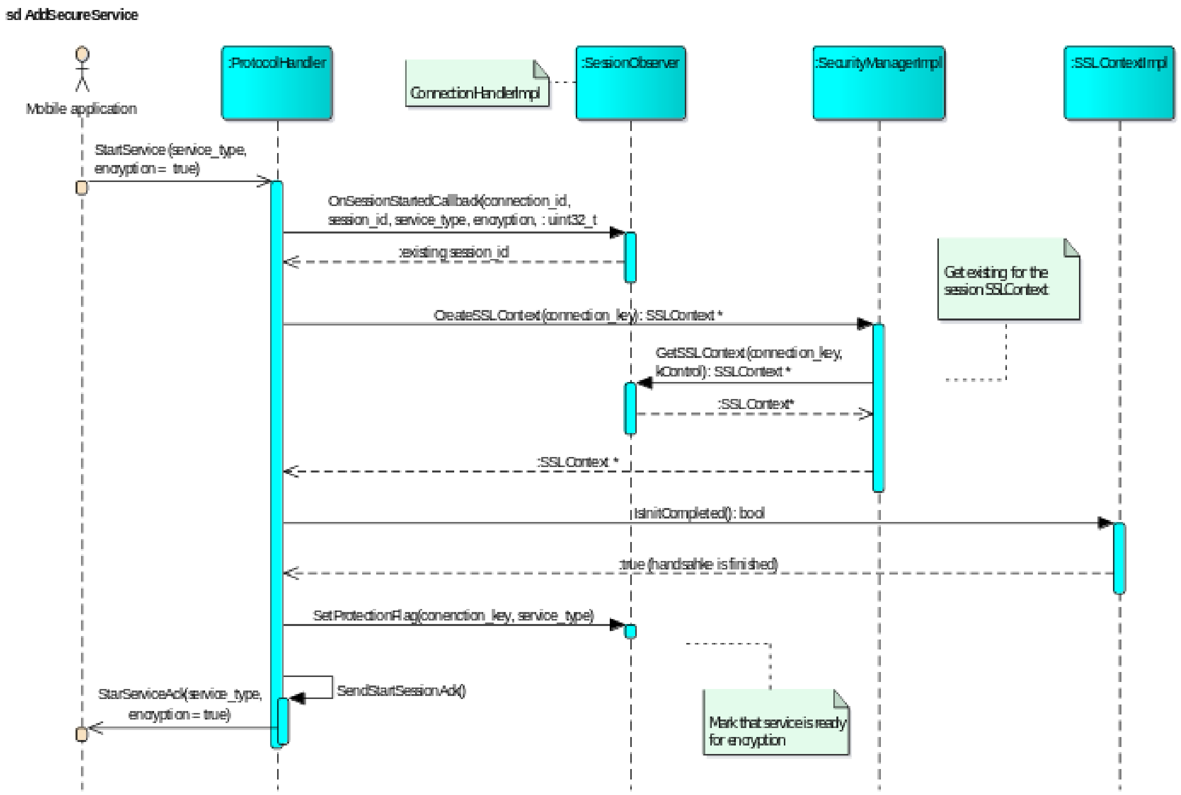
SECURITY MANAGER ADD ENCRYPTED SERVICE

To understand what the security manager does to start an encrypted service after it has been initialized, please take a look at this flow diagram.

SEQUENCE DIAGRAM

Security Manager Add Encrypted Service

View Diagram


sd AddSecureService
Mobile application
:ProtocolHandler
ConnectionHandlerImpl
:SessionObserver
:SecurityManagerImpl
:SSLContextImpl
StartService (service_type, encryption = true)
OnSessionStartedCallback(connection_id, session_id, service_type, encryption, : uint32_t
:existing session_id
CreateSSLContext(connection_key): SSLContext *
Get existing for the session SSLContext
GetSSLContext(connection_key, kControl): SSLContext *
:SSLContext*
:SSLContext *
IsInitCompleted(): bool
:true (handsahke is finished)
SetProtectionFlag(conenction_key, service_type)
SendStartSessionAck()
StarServiceAck(service_type, encryption = true)
Mark that service is ready for encryption

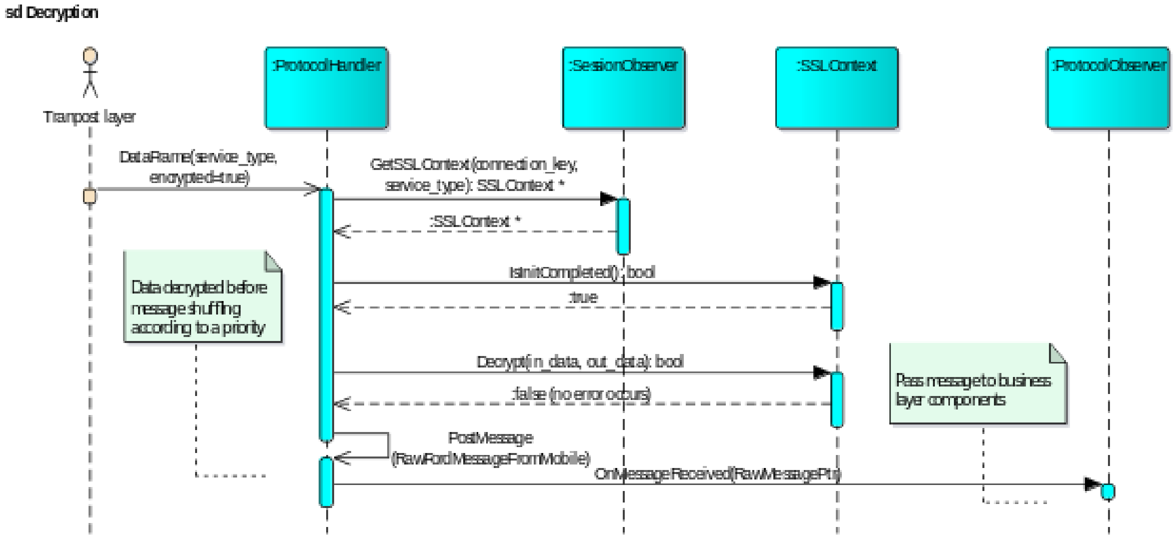
SECURITY MANAGER DECRYPT

To understand how the Security Manager decrypts data, please take a look at this flow diagram.

SEQUENCE DIAGRAM

Security Manager Decryption

View Diagram


sd Decryption
Tranpost layer
:ProtocolHandler
:SessionObserver
:SSLContext
:ProtocolObserver
DataFrame(service_type, encrypted=true)
GetSSLContext(connection_key, service_type): SSLContext *
:SSLContext *
IsInitCompleted() : bool
:true
Decrypt(in_data, out_data): bool
:false (no error occurs)
PostMessage (RawFordMessageFromMobile)
OnMessageReceived(RawMessagePtr)
Data decrypted before message shuffling according to a priority
Pass message to business layer components

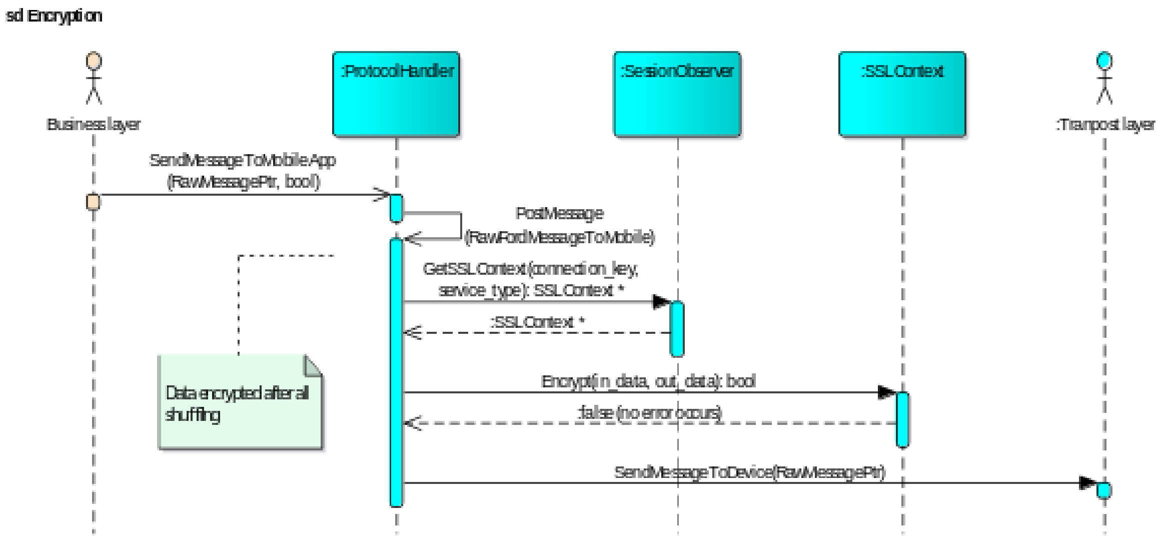
SECURITY MANAGER ENCRYPT

To understand how the Security Manager encrypts data, please take a look at this flow diagram.

SEQUENCE DIAGRAM

Security Manager Encryption

View Diagram


sd Encryption
Business layer
:ProtocolHandler
:SessionObserver
:SSLContext
:Tranport layer
SendMessageToMobileApp (RawMessagePtr, bool)
PostMessage (RawFordMessageToMobile)
GetSSLContext(connection_key, service_type): SSLContext *
:SSLContext *
Encrypt(in_data, out_data): bool
:false (no error occurs)
SendMessageToDevice(RawMessagePtr)
Data encrypted after all shuffling

SDL Logger

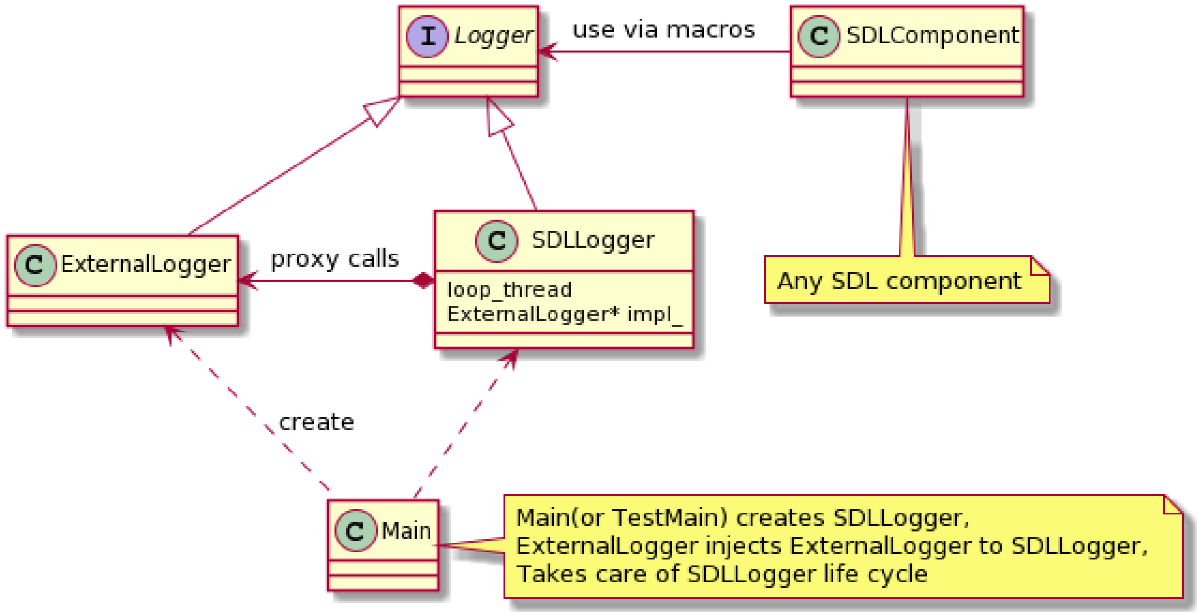
By default, SDL Core uses the log4cxx framework for logging.

By implementing logger abstraction, SDL now provides the capability to replace the log4cxx logger with any other logging package-such as boost or syslog.

SEQUENCE DIAGRAM

High Level Design

View Diagram



NOTE

All components have access to the `Logger` interface and use it for logging.

Logger Interface

Logger macros use the `Logger` interface for sending messages to the External Logger. The `Logger` interface contains only methods required by any SDL component to perform logging :

- instance() - singleton access
- PushLog(LogMessage)
- IsEnabledFor(LogLevel)
- DeInit()
- Flush()

Logger Implementation

`Logger` interface is implemented by `LoggerImpl`.

`LoggerImpl` uses the message loop thread to proxy log messages to a third party (external) logger.
`LoggerImpl` owns `ThirdPartyLoggerInterface` and controls it's lifetime.
`LoggerImpl` provides implementation of the singleton pattern.

Message loop thread in SDLLogger

The message loop thread is needed to avoid significant performance degradation at run time as logging calls are blocking calls and might take a significant amount of time. `LoggerImpl::PushLog` is a non-blocking call. It will put the log message into the queue and returns immediately.

If `ThirdPartyLoggerInterface` supports non blocking threaded logging, minor changes in `LoggerImpl` can be made with `use_message_loop_thread = false`.

Logger singleton

`Logger` is the only singleton class in SDL. The singleton pattern is required to access the logger instance from any component.

> **NOTE**
>
> `Logger::instance()` provides singleton by `Logger` interface. So SDL components do not have information about the logger implementation and the specific external logger.

Logger singleton with plugins

SDL plugins are shared libraries, so the `Logger` singleton could not be implemented with a Mayers singleton. A Mayers singleton would create an SDL logger instance for each plugin.

The idea is to pass a singleton pointer to each plugin during creation so that plugins can initialize the Logger::instance pointer with the instance received from SDL core.

SINGLETON INSTANCE IMPLEMENTATION

```
// ilogger.h
static Logger& instance(Logger* pre_init = nullptr);

...
// logger_impl.cc
Logger& Logger::instance(Logger* pre_init) {
  static Logger* instance_ = nullptr;
  if (pre_init) {
    assert(instance_ == nullptr);
    instance_ = pre_init;
  }
  assert(instance_);
  return *instance_;
}
```

`pre_init` is `nullptr` by default, so all components will access `instance_` static pointer for logging.
The `main()` function will need to create a `LoggerImpl` object and call `Logger::instance(logger implementation object)`;

PLUGIN IMPLEMENTATION

```
extern "C" PluginType* Create(Logger* logger_singleton_instance) {
  Logger::instance(logger_instance);
  return new PluginType();
}
```

SDL Core will pass a pointer to the logger singleton to the plugin so that the plugin shared library can initialize `Logger::instance` with the same pointer as the core portion.

Logger detailed design

Each source file creates `logger_` variable via macro `SDL_CREATE_LOG_VARIABLE`.
This variable is actually a string with the component name of the logger.
Some logger implementations (like log4cxx) may have separate severity or destination rules for each component.

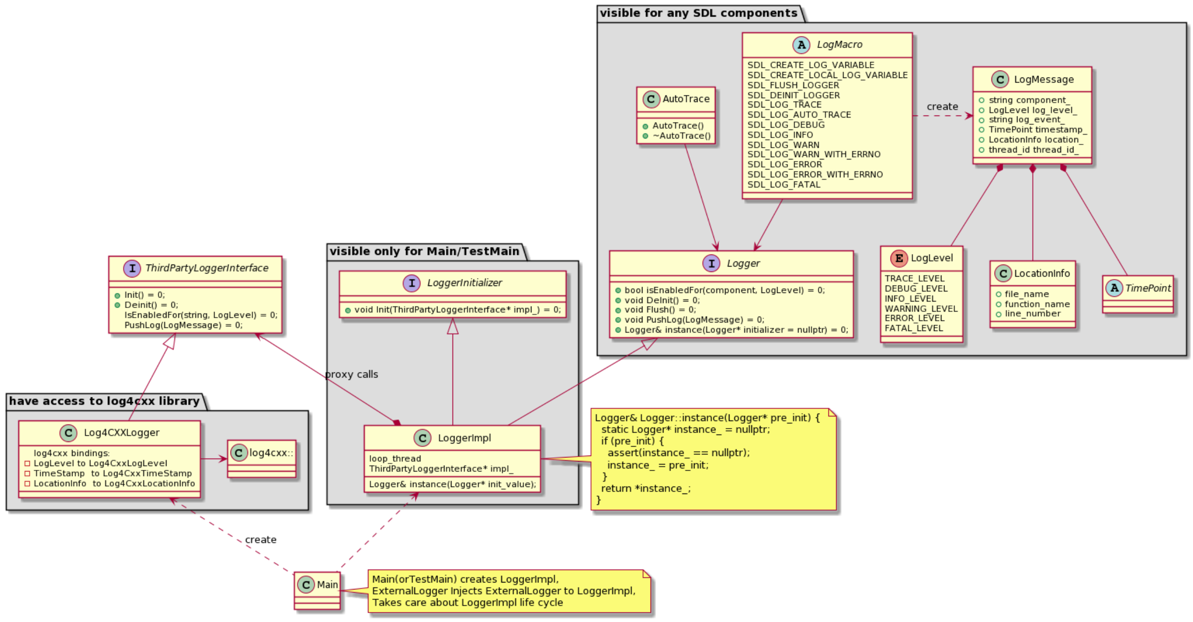
SDL implements all info required for log message :

- LogLevel enum
- Location info struct : location in the code
- TimePoint

SEQUENCE DIAGRAM

Detailed Design

View Diagram



```
Logger& Logger::instance(Logger* pre_init) {
  static Logger* instance_ = nullptr;
  if (pre_init) {
    assert(instance_ == nullptr);
    instance_ = pre_init;
  }
  return *instance_;
}
```

Main(orTestMain) creates LoggerImpl,
ExternalLogger Injects ExternalLogger to LoggerImpl,
Takes care about LoggerImpl life cycle

LoggerInitializer interface

`LoggerInitializer` specifies the interface required for `main()` to initialize the logger but is not required for any other SDL components.

`LoggerInitializer::Init` takes the third party logger implementation as an argument.
- `Init(std::unique_ptr<ThirdPartyLoggerInterface>&& third_party)`

ThirdPartyLogger interface

`ThirdPartyLoggerInterface` describes interfaces that should be implemented by the external logger adapter.

This interface should be inherited by external logger implementations.

Implementing another logger

To use another (not log4cxx) logger, you should:

- Create a class which inherits from the `ThirdPartyLoggerInterface` class

```
AnotherOneLoggerImpl : ThirdPartyLoggerInterface {
  void Init() override;
  void DeInit() override;
  void IsEnabledFor(LogLevel) override;
  void PushLog(const LogMessage& log_message) override;

  void SomeCustomMethod(parameters);
}
```

- Create an instance of the third party logger implementation(`AnotherOneLoggerImpl`) in `main()` and set it up for `LoggerImpl`.

NOTE

`Logger::instance` does not own the logger instance. The `main` function is responsible for the `sdl_logger_instance_` life-cycle.

```
// main.cpp
int main(argc, argv) {
    auto external_logger_ = std::make_unique<AnotherOneLoggerImpl>();
    external_logger_->SomeCustomMethod(argv);
    auto sdl_logger_instance_ = new LoggerImpl(std::move(external_logger_));
    Logger::instance(sdl_logger_instance_);
    sdl_logger_instance_->Init(std::move(external_logger_));
  // Futher application code may use Logger::instance() for logging

    delete sdl_logger_instance_;
}
```

Migrating SDL Core 6.1 to 7.0

The 7.0 release had a number of changes and additions to the HMI API that will require updates to your SDL Core integration in your head unit.

Environment Update

The minimum environment requirements have changed for Ubuntu 18. GCC Version 7.5.x is now recommended over the previously recommended GCC Version 7.3.x.

Breaking Changes

- The `url` parameter in the HMI API and RPC Spec has removed its max length requirement.
- The parameter `appID` has been removed from VehicleInfo.UnsubscribeVehicleData request to the HMI.

Newly Deprecated

Deprecated Character Sets

Character sets `TYPE2SET`, `TYPE5SET`, `CID1SET`, and `CID2SET` have been deprecated. These character sets only had proprietary significance and HMIs can now choose from the following character sets:

- ASCII
- ISO_8859_1
- UTF_8

Deprecated HMI RPC: OnFindApplications

This unimplemented RPC has been marked as deprecated and should be removed in the next major version change of SDL Core.

Deprecated Vehicle Data

Vehicle Data parameters `fuelLevel` and `fuelLevel_state` have been deprecated. Please make updates to use expanded vehicle data struct `FuelRange`.

```
<struct name="FuelRange">
  <param name="type" type="Common.FuelType" mandatory="false"/>
  <param name="range" type="Float" minvalue="0" maxvalue="10000" mandatory="false">
    <description>
      The estimate range in KM the vehicle can travel based on fuel level and consumption.
    </description>
  </param>
  <param name="level" type="Float" minvalue="-6" maxvalue="1000000" mandatory="false">
  <description>The relative remaining capacity of this fuel type (percentage).</description>
  </param>
  <param name="levelState" type="Common.ComponentVolumeStatus" mandatory="false">
    <description>The fuel level state</description>
  </param>
  <param name="capacity" type="Float" minvalue="0" maxvalue="1000000" mandatory="false">
  <description>The absolute capacity of this fuel type.</description>
  </param>
  <param name="capacityUnit" type="Common.CapacityUnit" mandatory="false">
  <description>The unit of the capacity of this fuel type such as liters for gasoline or kWh for batteries.</description>
  </param>
</struct>
```

Vehicle Data parameter `prndl` has been deprecated. Please make updates to use the new vehicle data struct `GearStatus`.

```
<struct name="GearStatus">
  <param name="userSelectedGear" type="Common.PRNDL" mandatory="false">
    <description>Gear position selected by the user i.e. Park, Drive,
Reverse</description>
  </param>
  <param name="actualGear" type="Common.PRNDL" mandatory="false">
    <description>Actual Gear in use by the transmission</description>
  </param>
  <param name="transmissionType" type="Common.TransmissionType"
mandatory="false">
    <description>Tells the transmission type</description>
  </param>
</struct>
```

Additions

Vehicle Data

- FuelRange was expanded to replace `fuelLevel` and `fuelLevel_state` parameters
- New vehicle data type: GearStatus to replace `prndl` parameter
- New vehicle data type: `StabilityControlStatus`
- New vehicle data type: `WindowStatus`
- New vehicle data type: `HandsOffSteering`

It is not required to implement all vehicle data types. If a type is unsupported by your headunit, please be sure to respond to SDL Core with result `UNSUPPORTED_RESOURCE` if an unsupported request has been made.

HMI UI Additions

MENU CHANGES

SDL Core 7.0 adds extended capabilities to the app menu. SDL Core now supports nested submenus, dynamic menus, and menu browsing limitations while driver distraction mode is enabled.

Nested Submenus:

An app can now request to add a submenu to another submenu by specifying a `parentID`. This did not require any new parameters on the HMI side. HMIs should be updated to process `parentID` in an `AddSubMenu` request. This param was previously only reserved for `AddCommands`.

```
<function name="AddSubMenu" messagetype="request">
...
    <param name="menuParams" type="Common.MenuParams" mandatory="true">
        <description>Position, parentID, and name of menu to be added.</description>
    </param>
...
```

Dynamic Menus:

Two new RPCs were added to HMI API: UI.OnUpdateFile and UI.OnUpdateSubmenu. UI.OnUpdateFile request allows the HMI to request images from an SDL connected app when needed in an effort to reduce the amount of data an app needs to save on the head unit. UI.OnUpdateSubmenu request allows the HMI to dynamically request when submenu information is populated by the app. This functionality helps reduce the system load when an app first connects as the app is not required to load all menu contents onto the head unit immediately.

```
<function name="OnUpdateFile" messagetype="notification">
   <description>For the HMI to tell Core that a file needs to be retrieved from the app.
</description>
   <param name="appID" type="Integer" mandatory="true">
   <description>ID of application related to this RPC.</description>
   </param>
   <param name="fileName" type="String" maxlength="255" mandatory="true">
   <description>File reference name.</description>
   </param>
</function>

<function name="OnUpdateSubMenu" messagetype="notification">
   <description>For the HMI to tell Core that a submenu needs
updating</description>

   <param name="appID" type="Integer" mandatory="true">
   <description>ID of application related to this RPC.</description>
   </param>

   <param name="menuID" type="Integer" minvalue="0" maxvalue="2000000000"
mandatory="true">
   <description>This menuID must match a menuID in the current menu
structure</description>
   </param>

   <param name="updateSubCells" type="Boolean" mandatory="false">
   <description>If not set, assume false. If true, the app should send AddCommands
with parentIDs matching the menuID. These AddCommands will then be attached to
the submenu and displayed if the submenu is selected.</description>
   </param>
</function>
```

Dynamic menus are optional, and the HMI's ability to support this feature is designated by the `DynamicUpdateCapabilities` struct.

```
<struct name="DynamicUpdateCapabilities">
    <param name="supportedDynamicImageFieldNames" type="ImageFieldName"
array="true" mandatory="false" minsize="1">
    <description>An array of ImageFieldName values for which the system supports
sending OnFileUpdate notifications. If you send an Image struct for that image field
with a name without having uploaded the image data using PutFile that matches that
name, the system will request that you upload the data with PutFile at a later point
when the HMI needs it. The HMI will then display the image in the appropriate field. If
not sent, assume false.</description>
    </param>

    <param name="supportsDynamicSubMenus" type="Boolean" mandatory="false">
    <description>If true, the head unit supports dynamic sub-menus by sending
OnUpdateSubMenu notifications. If true, you should not send AddCommands that
attach to a parentID for an AddSubMenu until OnUpdateSubMenu is received with the
menuID. At that point, you should send all AddCommands with a parentID that match
the menuID. If not set, assume false.</description>
    </param>
</struct>
```

Driver Distraction Limitations:

An HMI integration may choose to limit the amount of data available to the user when driver distraction is enabled. These limations include setting a limit on the number of menu items shown to the user in a given view, as well as setting a limit on how deep a user can drill down into nested submenus. This HMI capability is communicated to SDL Core and connected apps via the `DriverDistractionCapability` struct.

Setting these limits does not change the behavior of SDL Core. It is up to the HMI's integration to honor the designated limits and control how much information is available to the user.

```
<struct name="DriverDistractionCapability">
    <param name="menuLength" type="Integer" mandatory="false">
        <description>The number of items allowed in a Choice Set or Command menu
while the driver is distracted</description>
    </param>
    <param name="subMenuDepth" type="Integer" minvalue="1" mandatory="false">
        <description>The depth of submenus allowed when the driver is distracted. e.g.
3 == top level menu -> submenu -> submenu; 1 == top level menu only</description>
    </param>
</struct>
```

NEW UI COMPONENT: SUBTLE ALERT

`SubtleAlert` RPC was added as a less intrusive UI notification when compared to the `Alert` RPC. The `OnSubtleAlertPressed` notification was also added as a way for mobile apps to be aware of and optionally take action when a user clicks on a `SubtleAlert` notification.

```
<function name="SubtleAlert" messagetype="request">
   <description>Request from SDL to show a subtle alert message on the display.</description>
   <param name="alertStrings" type="Common.TextFieldStruct" mandatory="true" array="true" minsize="0" maxsize="2">
      <description>Array of lines of alert text fields. See TextFieldStruct. Uses subtleAlertText1, subtleAlertText2.</description>
   </param>
   <param name="alertIcon" type="Common.Image" mandatory="false">
      <description>
         Image to be displayed for the corresponding alert. See Image.
         If omitted, no (or the default if applicable) icon should be displayed.
      </description>
   </param>
   <param name="duration" type="Integer" mandatory="false" minvalue="3000" maxvalue="10000">
      <description>Timeout in milliseconds.</description>
   </param>
   <param name="softButtons" type="Common.SoftButton" mandatory="false" minsize="0" maxsize="2" array="true">
      <description>App defined SoftButtons</description>
   </param>
   <param name="alertType" type="Common.AlertType" mandatory="true">
      <description>Defines if only UI or BOTH portions of the Alert request are being sent to HMI Side</description>
   </param>
   <param name="appID" type="Integer" mandatory="true">
      <description>ID of application requested this RPC.</description>
   </param>
   <param name="cancelID" type="Integer" mandatory="false">
      <description>
         An ID for this specific alert to allow cancellation through the `CancelInteraction` RPC.
      </description>
   </param>
</function>

<function name="SubtleAlert" messagetype="response">
   <param name="tryAgainTime" type="Integer" mandatory="false" minvalue="0" maxvalue="2000000000">
      <description>Amount of time (in milliseconds) that SDL must wait before resending an alert. Must be provided if another system event or overlay currently has a higher priority than this alert.</description>
   </param>
</function>

<function name="OnSubtleAlertPressed" messagetype="notification">
   <description>
   Sent when the alert itself is touched (outside of a soft button). Touching (or otherwise selecting) the alert should open the app before sending this notification.
   </description>
   <param name="appID" type="Integer" mandatory="true">
```

```
        <description>ID of application that is related to this RPC.</description>
    </param>
</function>
```

Webengine Projection Support

A new `WEB_VIEW` `AppHMIType` and template layout was added which will allow apps to render a template-independent view in a browser environment with JavaScript and HTML.

```
<enum name="AppHMIType">
  <description>Enumeration listing possible app types.</description>
...
   <element name="WEB_VIEW" />
</enum>
```

This `AppHMIType` must be explicitly specified in an app's policy table entry for SDL Core to allow it to be used.

Policy Table Entry:

```
...
"app_policies": {
    "webengine_appID": {
+       "AppHMIType": ["WEB_VIEW"],
        "keep_context": false,
        "steal_focus": false,
        "priority": "NONE",
        "default_hmi": "NONE",
        "groups": [
            "Base-4"
        ],
        "RequestType": [],
        "RequestSubType": []
    },
...
```

New UI.GetCapabilities parameter: pcmStreamCapabilities

This parameter was added to the HMI API to align better with the Mobile API.

```
<function name="GetCapabilities" messagetype="response">
   <param name="displayCapabilities" type="Common.DisplayCapabilities"
mandatory="true">
       <description>Information about the capabilities of the display: its type, text field
supported, etc. See DisplayCapabilities. </description>
   </param>
   <param name="audioPassThruCapabilities"
type="Common.AudioPassThruCapabilities" mandatory="true"/>
   <param name="hmiZoneCapabilities" type="Common.HmiZoneCapabilities"
mandatory="true"/>
   <param name="softButtonCapabilities" type="Common.SoftButtonCapabilities"
minsize="1" maxsize="100" array="true" mandatory="false">
       <description>Must be returned if the platform supports on-screen SoftButtons.
</description>
   </param>
   <param name="hmiCapabilities" type="Common.HMICapabilities"
mandatory="false">
       <description>Specifies the HMI’s capabilities. See HMICapabilities.
</description>
   </param>
   <param name="systemCapabilities" type="Common.SystemCapabilities"
mandatory="false">
       <description>Specifies system capabilities. See
SystemCapabilities</description>
   </param>
+   <param name="pcmStreamCapabilities"
type="Common.AudioPassThruCapabilities" mandatory="false"/>
</function>
```

Migrating SDL Core 7.0 to 7.1

The 7.1 release had a number of changes and additions to the HMI API that will require updates to your SDL Core integration in your head unit.

Environment Update

The default supported version was changed to Ubuntu 20. Recommended GCC Version 9.3.x.

Support was added for OpenSSL 1.1, we recommend updating your version of the library accordingly.
Along with support for OpenSSL 1.1, a configurable `SecurityLevel` field was added to the INI file. This value can be customized depending on the security requirements of your system (see the OpenSSL documentation for a description of each security level)

Newly Deprecated

Deprecated SyncPData RPCs

- RPC request and response for `EncodedSyncPData` has been marked as deprecated.
- RPC notification `OnEncodedSyncPData` has been marked as deprecated.

Deprecated UI params

- `TextFieldName` element `mediaClock` has been marked as deprecated.
- `Show` RPC param `mediaClock` has been marked as deprecated.
- `RegisterAppInterface` parameters `vehicleType` and `systemSoftwareVersion` has been marked as deprecated. Please make updates to use the parameters from the `StartService ACK` protocol message.

Deprecated Functions

- The function `DynamicApplicationData::IsSubMenuNameAlreadyExist` has been marked as deprecated and should be removed in the next major version change of SDL Core. Please make updates to remove all uses of the function.
- The function `ApplicationManagerImpl::OnAppStreaming(uint32_t, protocol_handler::ServiceType, const Application::StreamingState)` has been marked as deprecated and should be removed in the next major version change of SDL Core. Please make

updates to use the new function signature `ApplicationManagerImpl::OnAppStreaming(uint32_t, protocol_handler::ServiceType, bool)`.

- The function `ProtocolHandlerImpl::NotifySessionStarted(const SessionContext&, std::vector<std::string>&, const std::string)` has been marked as deprecated and should be removed in the next major version change of SDL Core. Please make updates to use the new function signature `ProtocolHandlerImpl::NotifySessionStarted(SessionContext&, std::vector<std::string>&, const std::string)`.
- The function `file_system::ConvertPathForURL` has been marked as deprecated and should be removed in the next major version change of SDL Core. Please make updates to remove all uses of the function.

Deprecated Vehicle Data

- Vehicle Data parameter `externalTemperature` has been deprecated. Please make updates to use the new vehicle data struct `climateData`.
- Vehicle Data parameters `driverDoorAjar`, `passengerDoorAjar`, `rearLeftDoorAjar` and `rearRightDoorAjar` have been deprecated. Please make updates to use the new `doorStatuses` parameter.

Additions

Vehicle Data

- `BodyInformation` was expanded to replace `driverDoorAjar`, `passengerDoorAjar`, `rearLeftDoorAjar` and `rearRightDoorAjar` parameters.
- New vehicle data type: `climateData` to replace `externalTemperature` parameter.
- New vehicle data type: `seatOccupancy`.

It is not required to implement all vehicle data types. If a type is unsupported by your headunit, please be sure to respond to SDL Core with the result `UNSUPPORTED_RESOURCE` if an unsupported request has been made.

HMI UI Additions

CUSTOM PLAYBACK RATES FOR SETMEDIACLOCKTIMER

A media app now has the ability to specify a custom playback rate (ex. 125% speed) when setting the media playback timer and progress bar.

Added new parameter `countRate` to the `SetMediaClockTimer` RPC

```
<function name="SetMediaClockTimer" functionID="SetMediaClockTimerID"
messagetype="request" since="1.0">
    <description>Sets the initial media clock value and automatic update method.
</description>

    <!-- New Parameter -->
    <param name="countRate" type="Float" minvalue="0.1" maxvalue="100.0"
defvalue="1.0" mandatory="false">
        <description>
        The value of this parameter is the amount that the media clock timer will
advance per 1.0 seconds of real time.

        Values less than 1.0 will therefore advance the timer slower than real-time,
while values greater than 1.0 will advance the timer faster than real-time.

        e.g. If this parameter is set to `0.5`, the timer will advance one second per two
seconds real-time, or at 50% speed. If this parameter is set to `2.0`, the timer will
advance two seconds per one second real-time, or at 200% speed.
        </description>
    </param>
</function>
```

MEDIA SKIP INDICATORS

A media app now has the ability to change the indicators for the `SEEKLEFT` and `SEEKRIGHT` buttons to show either time skip buttons or track skip buttons.

- Added new parameters `forwardSeekIndicator` and `backSeekIndicator` to the `SetMediaClockTimer` RPC.

```
<enum name="SeekIndicatorType">
   <element name="TRACK">
   <element name="TIME">
</enum>

<struct name="SeekStreamingIndicator">
   <description>
      The seek next / skip previous subscription buttons' content
   </description>

   <param name="type" type="SeekIndicatorType" mandatory="true" />
   <param name="seekTime" type="Integer" minvalue="1" maxvalue="99"
mandatory="false">
      <description>If the type is TIME, this number of seconds may be present
alongside the skip indicator. It will indicate the number of seconds that the currently
playing media will skip forward or backward.</description>
   </param>
</struct>

<function name="SetMediaClockTimer" messagetype="request">
  <!– Additions –>
  <param name="forwardSeekIndicator" type="SeekStreamingIndicator"
mandatory="false" />
  <param name="backSeekIndicator" type="SeekStreamingIndicator"
mandatory="false" />
</function>
```

MAIN MENU UI UPDATES

SDL Core 7.1 adds extended capabilities to the `AddSubMenu` and `AddCommand` RPCs. Both `AddSubmenu` and `AddCommand` now have additional optional textfields as well as an optional secondary image.

AddSubmenu:

```
<function name="AddSubMenu" functionID="AddSubMenuID"
messagetype="request">
   <description>Adds a sub menu to the in-application menu.</description>

   <!– New Parameters –>
   <param name="secondaryText" maxlength="500" type="String" mandatory="false">
      <description>Optional secondary text to display</description>
   </param>
   <param name="tertiaryText" maxlength="500" type="String" mandatory="false">
      <description>Optional tertiary text to display</description>
   </param>
   <param name="secondaryImage" type="Image" mandatory="false">
      <description>Optional secondary image struct for sub-menu cell</description>
   </param>
</function>
```

AddCommand:

```
<function name="AddCommand" functionID="AddCommandID"
messagetype="request">
   <description>
      Adds a command to the in application menu.
      Either menuParams or vrCommands must be provided.
   </description>

   <!– New Parameters –>
   <param name="secondaryImage" type="Image" mandatory="false">
      <description>Optional secondary image struct for menu cell</description>
   </param>
</function>

<struct name="MenuParams" since="1.0">
   <!– New Parameters –>
   <param name="secondaryText" maxlength="500" type="String" mandatory="false">
      <description>Optional secondary text to display</description>
   </param>
   <param name="tertiaryText" maxlength="500" type="String" mandatory="false">
      <description>Optional tertiary text to display</description>
   </param>
</struct>
```

BROADENING CHOICE UNIQUENESS

Prior to SDL Core 7.1, choice set choices and menu commands were required to have unique primary text. SDL Core 7.1 removes this restriction.

KEYBOARD ENHANCEMENTS

SDL Core 7.1 adds a new `NUMERIC` keyboard layout and new enhancements to allow apps to mask entered characters and change special characters shown on the keyboard layout.

OEM exclusive apps support

SDL Core 7.1 adds the ability to share vehicle type information before sending the Register App interface request. This will enable SDL adopters to provide exclusive apps to their users depending on vehicle type

The vehicle type information parameters have been added to the BSON payload of the `StartServiceACK` protocol message

TAG NAME	TYPE	DESCRIPTION
make	String	Vehicle make
model	String	Vehicle model
modelYear	String	Vehicle model year
trim	String	Vehicle trim
systemSoftwareVersion	String	Vehicle system software version
systemHardwareVersion	String	Vehicle system hardware version

The vehicle type information parameters (`vehicleType` and `systemSoftwareVersion`) in `RegisterAppInterface` have been deprecated in favor of these additions

Video streaming capability updates

PREFERRED FPS

- Added new parameter `preferredFPS` to the `VideoStreamingCapability` struct.

```
<struct name="VideoStreamingCapability" since="4.5">
    <description>Contains information about this system's video streaming
capabilities.</description>
    ...
    <!– new param –>
    <param name="preferredFPS" type="Integer" minvalue="0" maxvalue="2147483647"
mandatory="false">
        <description>The preferred frame rate per second of the head unit. The mobile
application / app library may take other factors into account that constrain the frame
rate lower than this value, but it should not perform streaming at a higher frame rate
than this value.</description>
    </param>
</struct>
```

UPDATING VIDEO STREAMING CAPABILITIES DURING IGNITION CYCLE

SDL Core 7.1 adds the ability for an application to update its video streaming capabilities during the ignition cycle. This will allow SDL to handle uses cases that require dynamic resolution switching (Picture-in-Picture, preview, split-screen, etc.)

- Added new parameter `additionalVideoStreamingCapabilities` to the `VideoStreamingCapability` struct.

```
<struct name="VideoStreamingCapability" since="4.5">
    <!– Existing params –>
    <param name="additionalVideoStreamingCapabilities"
type="VideoStreamingCapability" array="true" minvalue="1" maxvalue="100"
mandatory="false" since="7.1">
    </param>
</struct>
```

- Added new RPC notification `OnAppCapabilityUpdated` which can be sent by an app, as well as related structs `AppCapability` and `AppCapabilityType`.

```
<function name="OnAppCapabilityUpdated" functionID="OnAppCapabilityUpdatedID"
messagetype="notification" since="7.1">
   <description>A notification to inform SDL Core that a specific app capability has
changed.</description>
   <param name="appCapability" type="AppCapability" mandatory="true">
      <description>The app capability that has been updated</description>
   </param>
</function>

<struct name="AppCapability" since="7.1">
   <param name="appCapabilityType" type="AppCapabilityType" mandatory="true">
      <description>Used as a descriptor of what data to expect in this struct. The
corresponding param to this enum should be included and the only other param
included.</description>
   </param>
   <param name="videoStreamingCapability" type="VideoStreamingCapability"
mandatory="false">
      <description>Describes supported capabilities for video streaming
</description>
   </param>
</struct>

<enum name="AppCapabilityType" since="7.1">
   <description>Enumerations of all available app capability types</description>
   <element name="VIDEO_STREAMING"/>
</enum>
```

Migrating SDL Core 7.1 to 8.0

Environment Updates

Ubuntu Versions

SDL Core 8.0.0 no longer supports Ubuntu 16. Supported versions of SDL Core are Ubuntu 18.04 and Ubuntu 20.04.

SSL Versions

SDL Core 8.0.0 dropped support for libssl1.0. Developers should install `libssl-dev` instead of `libssl1.0-dev`.

Updates to CMAKE Build Configuration

Removed Flag `ENABLE_HMI_PTU_DECRYPTION`

`ENABLE_HMI_PTU_DECRYPTION` was removed from the build configuration. Behaviors defined by ON/OFF options are now both supported without the need for this build flag.

Boost Logger

The default logger is still using LOG4CXX but the option is now available to use Boost for the logger. When porting SDL Core to different Linux environments, the LOG4CXX logger was known to cause dependency issues. Boost is offered as an alternative logger in hopes of making porting SDL Core to different environments easier.

```
set(LOGGER_NAME "LOG4CXX" CACHE STRING "Logging library to use (BOOST,
LOG4CXX)")
```

Updates to Configuration File smartDeviceLink.ini

DefaultTimeoutCompensation

This parameter was added to the smartDeviceLink.ini configuration to compensate for transfer and processing time of requests. This value is added to the `DefaultTimeout` parameter when calculating the RPC request timeout. Previously, specific requests such as Alert were hardcoded to extend their default timeout, now timeout compensation is configurable and applied to all requests.

```
; Extra time to compensate default timeout due to external delays
DefaultTimeoutCompensation = 1000
```

Icons Storage Folder

The default parameter `AppIconsFolder` was updated to use a directory named “icons”. This value used to be “storage”.

```
; Specify a dedicated folder, as old files in this folder can be automatically removed
AppIconsFolder = icons
```

HMI Behavior Changes

Avoid Custom Button Subscription in Case HMI Incompatibility

SDL Core 8.0.0 no longer automatically subscribes to `CUSTOM_BUTTON`. If an HMI supports soft buttons, it must include an entry for `CUSTOM_BUTTON` in its button capabilities in order for mobile to receive `OnButtonPress` and `OnButtonEvent` notifications.

OnEventChanged (PHONE_CALL)

The behavior of the `PHONE_CALL` event was changed to only affect the `audioStreamingState` of an app. Rather than automatically deactivating the active app, SDL Core will now only change the `audioStreamingState` of all apps to `NOT_AUDIBLE` when `BC.OnEventChanged(PHONE_CALL, active=true)` is sent, leaving each app's `hmiLevel` unchanged. This allows the HMI to start a phone call in the background without leaving the app screen, if desired.

The HMI can still control the `hmiLevel` of the app during a phone call event by sending `BC.OnAppDeactivated(appID)` and `BC.OnAppActivated(appID)` where appropriate.

HMI API Updates

Buttons.SubscribeButton

`Buttons.OnButtonSubscription` notification was replaced by `Buttons.SubscribeButton` request and response.

```
<function name="SubscribeButton" messagetype="request">
    <description>
      Subscribes to buttons.
    </description>

    <param name="appID" type="Integer" mandatory="true">
      <description>The ID of the application requesting this button subscription.
</description>
    </param>

    <param name="buttonName" type="ButtonName" mandatory="true">
      <description>Name of the button to subscribe.</description>
    </param>
</function>

<function name="SubscribeButton" messagetype="response"> </function>
```

Buttons.UnsubscribeButton

`Buttons.UnsubscribeButton` request and response were added to allow SDL Core to request that the HMI unsubscribes an application from a specific button.

```
<function name="UnsubscribeButton" messagetype="request">
    <description>
      Unsubscribes from buttons.
    </description>

    <param name="appID" type="Integer" mandatory="true">
      <description>The ID of the application requesting this button unsubscription.
</description>
    </param>

    <param name="buttonName" type="ButtonName" mandatory="true">
      <description>Name of the button to unsubscribe.</description>
    </param>
  </function>

<function name="UnsubscribeButton" messagetype="response"></function>
```

Restructuring OnResetTimeout

`UI.OnResetTimeout` and `TTS.OnResetTimeout` were removed in place of using a broader RPC, `BasicCommunication.OnResetTimeout`.

This updated `OnResetTimeout` RPC can be used across all interfaces for all request functions.

The parameters in the notification have also changed:
- The parameter `requestID` is used instead of `appID` to identify which specific request should have its timeout extended.
- The parameter `methodName` should include the interface name and the RPC. For example: `”TTS.Speak”`.
- The parameter `resetPeriod` allows the HMI to specify how long Core should delay the application request’s timeout.

```
<interface name="BasicCommunication">
...
<function name="OnResetTimeout" messagetype="notification" since="X.Y">
   <description>
      HMI must send this notification to SDL for method instance for which timeout
needs to be reset
   </description>
   <param name="requestID" type="Integer" mandatory="true">
      <description>
         Id between HMI and SDL which SDL used to send the request for method in
question, for which timeout needs to be reset.
      </description>
   </param>
   <param name="methodName" type="String" mandatory="true">
      <description>
         Name of the function for which timeout needs to be reset
      </description>
   </param>
   <param name="resetPeriod" type="Integer" minvalue="0" maxvalue="1000000"
mandatory="false">
      <description>
         Timeout period in milliseconds, for the method for which timeout needs to be
reset.
         If omitted, timeout would be reset by defaultTimeout specified in
smartDeviceLink.ini
      </description>
   </param>
</function>
…
</interface>
```

Migrating SDL Core 8.0 to 8.1

API changes

The 8.1 release had a few changes to the HMI API that will require updates to your SDL Core integration in your head unit.

Removal of UI.SetDisplayLayout

With the release of SDL Core 8.1, the `UI.SetDisplayLayout` RPC has been removed from the HMI API.

```
...
- <function name="SetDisplayLayout" messagetype="request">
-   <description>This RPC is deprecated. Use Show RPC to change layout.</description>
-   <param name="displayLayout" type="String" maxlength="500" mandatory="true">
-     <description>
-       Predefined or dynamically created screen layout.
-       Currently only predefined screen layouts are defined.
-     </description>
-   </param>
-   <param name="appID" type="Integer" mandatory="true">
-     <description>ID of application related to this RPC.</description>
-   </param>
-   <param name="dayColorScheme" type="Common.TemplateColorScheme" mandatory="false"></param>
-   <param name="nightColorScheme" type="Common.TemplateColorScheme" mandatory="false"></param>
- </function>

- <function name="SetDisplayLayout" messagetype="response">
-   <description>This RPC is deprecated. Use Show RPC to change layout.</description>
-   <param name="displayCapabilities" type="Common.DisplayCapabilities" mandatory="false">
-     <description>See DisplayCapabilities</description>
-   </param>
-   <param name="buttonCapabilities" type="Common.ButtonCapabilities" minsize="1" maxsize="100" array="true" mandatory="false">
-     <description>See ButtonCapabilities</description >
-   </param>
-   <param name="softButtonCapabilities" type="Common.SoftButtonCapabilities" minsize="1" maxsize="100" array="true" mandatory="false">
-     <description>If returned, the platform supports on-screen SoftButtons; see SoftButtonCapabilities.</description >
-   </param>
-   <param name="presetBankCapabilities" type="Common.PresetBankCapabilities" mandatory="false">
-     <description>If returned, the platform supports custom on-screen Presets; see PresetBankCapabilities.</description >
-   </param>
- </function>
...
```

When an app sends a `SetDisplayLayout` request, SDL now transforms it into a `UI.Show` request (with the `templateConfiguration` parameter set based on the parameters defined in the `SetDisplayLayout` request) and forwards it to the HMI. The `UI.SetDisplayLayout` implementation was also removed from the SDL HMI and Generic HMI. However, developers may decide to keep their implementation to support older versions of SDL Core.

Removal of duplicate parameter from BasicCommunication.OnPutFile

The duplicate parameter `FileName` was removed from the `BasicCommunication.OnPutFile` RPC in the HMI API

```
<function name="OnPutFile" messagetype="notification" >
    <description>
      Notification that is sent to HMI when a mobile application uploads a file
    </description>
    ...
-   <param name="FileName" type="String" maxlength="255" mandatory="true">
-     <description>File reference name.</description>
-   </param>

    <param name="syncFileName" type="String" maxlength="255" mandatory="true">
      <description>File reference name.</description>
    </param>
    <param name="fileType" type="Common.FileType" mandatory="true">
        <description>Selected file type.</description>
    </param>
...
```

The parameter was unused. SDL Core uses `syncFileName` in the notification sent to the HMI.

Core behavior changes

Reject PROPRIETARY/HTTP SystemRequests when PTU is not in progress

With the release of 8.1, SDL Core will now reject incoming `PROPRIETARY` / `HTTP` SystemRequests when a policy table update (PTU) is not in progress and if an application not selected for the PTU sends the request.

This was identified as a security flaw since it would allow any application to trigger a PTU. For more information please see proposal 0337.

Doxygen Inline Documentation

For more detailed documentation on SDL Core, please visit the Doxygen webpage!

SDL Core FAQ

Here are a few of the most common questions new developers have around the SDL Core project.

What OS should I use to get started?

Currently the SDL Core repo is built for Ubuntu 20.04 as our default environment.

I'm getting a lot of compilation errors, how do I get past them?

The most common errors come from dependencies issues. Ensure your system has all the required packages to compile the project. Try running the commands in the dependencies section of the Getting Started guide.

Can I use SDL on Android OS?

There is no official port at the moment, so individual investigation will need to be done. Even though SDL is designed to work on most Linux systems, modifications might need to be made to the project to get it to work with your setup.

Why are my RPC requests being DISALLOWED by SDL?

The `DISALLOWED` result code is related to the RPC not being authorized in SDL Core's local policy table. Policy permissions for an app are added either in the preloaded policy table or through a policy table update.

The Policies Overview page provides general information about policies - explaining what they are used for, how the policy table gets updated, and how these updates are triggered. Policy Table Fields and App Policies go into more detail about the policy table structure and how to correctly add policy permissions for an application.

Changes I made to my preloaded policy table aren't reflected in SDL Core; what should I do?

If you are not running SDL Core for the first time, SDL Core will use the existing policy table database(`policy.sqlite`) which is stored in the build folder under `bin/storage/`. To make SDL Core parse the preloaded policy table again you have to delete the existing policy table database. In the bin folder run:

```
rm storage/policy.sqlite
```

Can I build SDL with/without certain features (such as logging or build tests)?

You can enable/disable certain features by modifying the CMakeLists.txt file. The CMake Build Configuration section contains a list of features which can be included/excluded for a build.

What options can I modify in SDL without having to rebuild?

The SmartDeviceLink.ini file located in your `build/src/appMain` directory is where runtime options can be configured for your instance of SDL Core. The INI Configuration page has more information about individual runtime options.

I'm experiencing choppy audio through Bluetooth; what should I do?

The default SDL Core repo actually performs an SDP on loop. Because SDP queries are a resource intensive operation it can cause the audio coming from the phone to become very choppy. This can be fixed by doing the following:

First, navigate to this line that reads:

```
: TransportAdapterImpl(new BluetoothDeviceScanner(this, true, 0),
```

Change it to:

```
: TransportAdapterImpl(new BluetoothDeviceScanner(this, false, 0),
```

That will cause the SDP queries to not be performed by default. This means you will need to create a way to perform SDP queries using an event trigger. So in the HMI implementation you will need to tie an event (button press or voice command) to sending the following RPC message to the Core service:

```
return ({
    'jsonrpc': '2.0',
    'method': 'BasicCommunication.OnStartDeviceDiscovery'
})
```

What is the integration time of SDL in an infotainment system?

Timing is dependent on the OEM or Supplier implementing SDL, and also dependent on factors such as OS, hardware, etc.